UNIVERSIDADE FEDERAL DE MINAS GERAIS

RAPHAEL RODRIGUES CAMPOS

Rank Aggregation Problem

Trabalho apresentado à disciplina de Projeto e Análise de Algoritmos do curso de pós-graduação em Ciência da Computação do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais.

Belo Horizonte
30 de Junho de 2015

Conteúdo

1 Introdução

Um problema clássico em análise de *ranks*, que tem se tornado primordial em inúmeras aplicações, é o ***rank aggregation problem***. Por exemplo, em ferramentas de *meta-search* na Web, que fazem consultas em máquinas de busca distintas, e então tentam produzir um resultado mais relevante a busca a partir da combinação dos *rankings* retornados pelas máquinas de busca, através de similaridades e diferenças entre eles. *Rank aggregation* tem várias aplicações em áreas como teoria da escolha social e ciência da computação, tais como meta-search (*previamente elucidado*), similaridade de busca e classificação[8]. ***Rank aggregation problem*** pode ser definido como: *o problema de encontrar um consenso, dadas as preferências de classificação de vários juízes*[5].

Rank aggregation tem sido estudado extensivamente no contexto de teoria da escolha social, onde vários "paradoxos de votação"foram descobertos. O ***consenso de Kemeny*** é um dos mais clássicos e naturais. Informalmente, dado listas ordendas $\tau_1, \tau_2, \tau_3, \ldots \tau_k$ com alternativas $\{1, 2, ..., n\}$, uma ordem ótima de Kemeny σ minimiza a soma das distâncias*"bubble sort"*[6], também conhecida como distância de Kendall tau[1]:

$$\min \sum_{i=1}^{k} K(\sigma, \tau_i) \tag{1}$$

Para ficar mais claro, $K(\sigma, \tau)$ pode ser calculado como o número de inversões da lista τ tendo em vista a ordem da lista σ, usando um algoritmo dividir para conquistar pode-se computar a distância de Kendall-Tau em $O(n \log n)$[7]. Assim, podemos observar, que a ótima de Kemeny produz a melhor ordem afim de minimizar o número de inversões em todas as listas. Infelizmente, a regra de Kemeny (1) é NP-Difícil[4].

1.1 O trabalho

Nesse trabalho é apresentado uma heurística baseada em um algoritmo guloso para encontrar o caminho hamiltoniano em um torneio, esse algoritmo é baseado na ideia de ordenação topológica em DAG e pode ser implementado em $O(|V| + |E|)$. Baseado nos algoritmos aproximados que serão apresentados na Seção 2.2, propõem-se uma heurística que executa os algoritmos aproximados e heurísticas, e então retorna o melhor resultado dentre eles. No intuito de melhorar a qualidade da solução, baseado no fato que cada algoritmo obtém resultados melhores dependendo da instância de entrada.

O trabalho é dividido nas seguintes seções: Seção 2, onde é mostrado uma visão geral sobre algumas heurísticas e algoritmos aproximados existentes para resolução do Rank Aggregation Problem. Na Seção 3 é apresentado um algoritmo exato para encontrar um consenso de Kemeny, o conceito de Kemenyzação Local e algumas heurísticas que solucionam o problema através da modelagem em grafos. Na Seção 4 são apresentadas as heurísticas propostas para *Rank Aggregation Problem*. Na seção 5 são apresentados os resultados experimentais e na seção 6 as conclusões do trabalho.

2 Trabalhos Relacionados

Como mencionado anteriormente, Rank Aggregation Problem tem sido extensivamente estudado no contexto de teoria da escolha social e ciência da computação, sobretudo nas áreas de meta-search, similaridade de busca e classificação. Rank aggregation torna-se ainda mais interessante no contexto da web onde temos algumas características particulares, tais como: as máquinas de buscas só fazem *crawling* de um subconjunto da web, portanto, cada máquina de busca pode retornar resultados distintos para uma mesma busca. Ademais, as máquinas de busca retornam apenas as m mais bem classificadas páginas para uma dada busca.

Para melhor elucidar o problema, temos como entrada n candidatos (ex. páginas web) e k eleitores (ex. máquinas de busca) e cada eleitor possui uma lista (parcial) de candidatos em ordem de preferência. E portanto, o algoritmo deve retornar um bom consenso na ordem dos candidatos. Como fora dito previamente na seção 1, o consenso de Kemeny é ótimo e também é a única função de preferência que é natural, consistente e Condorcet [9], porém é NP-difícil.

Eleitor	Primeira preferência	Segunda preferência	Terceira preferência
Eleitor 1	A	B	C
Eleitor 2	B	C	A
Eleitor 3	C	A	B

Tabela 2.1: Exemplo do paradoxo de Condorcet

Vários sistemas eleitorais foram propostos para esse problema secular, a seguir são mostrados, brevemente, alguns deles.

Na eleição por ordem de mérito, proposto por Jean-Charles Borda em 1770, definem-se valores para as posições dos candidatos em cada lista de preferência. Por exemplo, primeiro lugar vale um ponto, segundo lugar dois pontos e assim por diante. A pontuação do candidato é o somatório dos pontos, portanto, o vencedor é o candidato com menor pontuação.

Na proposta de Condorcet, proposto em 1785 por Marie J. A. N. Caritat, Marquês de Condorcet, particiona-se o conjunto de candidatos em dois, A e B. Se toda $a \in A$ e $b \in B$, a maioria dos eleitores prefere a a b então a agregação deve por todos elementos em A a frente de todos os elementos de B. O vencedor de Condorcet é o candidato que derrota todos os outros candidatos em comparação pareada. Todavia, o vencedor de Condorcet pode não existir e quando isso ocorre, é porque as maiorias conflitantes são compostas por indivíduos de grupos distintos, a Tabela 2.1 mostra um exemplo onde não há vencedor de Condorcet. Esse evento é conhecido como paradoxo de Condorcet.

Na proposta de Kemeny, feita em 1959, calcula-se a distância entre duas listas de preferência, onde as distâncias são calculadas contando o número de discordância na ordem de duas listas de preferêcia. Obtém-se a ordem dos candidatos que é a menos distante das ordens individuais.

2.1 Métricas de distância

Como fora dito na seção 1, encontrar a ordem σ que minimiza as distâncias Kendall tau é NP-difícil. Porém utilizando outras métricas pode-se encontrar uma ordem em tempo polinomial que seja garantidamente próximo a σ. Nessa subseção serão apresentadas algumas métricas com um pouco mais de detalhes.

Seja U o universo e $S \subset U$. E seja $\sigma = \{x_1 \geq x_2 \geq x_3\}$ e $\tau = \{x_2 \geq x_1 \geq x_3\}$, onde $x_i \in S$, duas listas de preferência (*rankings*). Pode haver três situações de listas:

1. Lista completa : $|S| = |\tau| = |U|$
2. Lista parcial: $S \subset U$, ex. máquinas de busca indexam apenas uma parte da web
3. Top m lista: um caso especial da lista parcial
   (a) Apenas o subconjunto S está ordenado
   (b) m primeiros mais bem classificados em S e estão sempre a frente dos que não estão na lista parcial.
   (c) ex. lista com m = 2, $\tau = \{x_2 \geq x_1\}$ (x_3 não está entre as 2 primeiras de τ)

2.1.1 Distância Kendall tau

A distância Kendall tau é dada pela equação 2:

$$K(\sigma, \tau) = \frac{|\{(i,j)|i < j, \sigma(i) < \sigma(i) \wedge \tau(i) > \tau(i)\}|}{\binom{|S|}{2}} \tag{2}$$

onde $|\{(i,j)|i < j, \sigma(i) < \sigma(i) \wedge \tau(i) > \tau(i)\}|$ pode ser calculado como o número de inversões da lista τ tendo em vista a ordem da lista σ. Usando um algoritmo dividir para conquistar baseado no mergesort pode-se computar a distância de Kendall-Tau eficientemente em $O(n \log n)$[7].

2.1.2 Distância Spearman footrule

A distância Spearman footrule é dada pela equação 3:

$$F(\sigma,\tau) = \frac{2\sum_{i=1}^{|S|} |\sigma(i) - \tau(i)|}{|S|^2} \tag{3}$$

onde $\sigma(i)$ e $\tau(i)$ retornam a posição do elemento i na lista σ e τ respectivamente.

2.1.3 Distâncias para listas parciais

As distâncias Spearman footrule e Kendall tau são métricas e, portanto, podem ser facilmente estendidas para várias listas [5].

$$Distancia(\sigma,\tau_1,\tau_2,...,\tau_k) = \frac{\sum_{i=1}^{k} Distancia(\sigma,\tau_i)}{k} \tag{4}$$

A extesão das métricas para listas parciais é dada por :

$$Distancia(\sigma,\tau_1,\tau_2,...,\tau_k) = \frac{\sum_{i=1}^{k} Distancia(\sigma_{|\tau_i},\tau_i)}{k} \tag{5}$$

Onde o universo U é a união entre os elementos em τ_i e σ é uma lista completa. Antes de computar a distância entre σ e τ_i, τ_i é projetado em σ. Esta projeção é representada por $\sigma_{|\tau_i}$ na equacão 5 [5].

2.2 Heurísticas e Algoritmos aproximados na literatura

Na literatura há várias abordagens para encontrar um consenso próximo ao ótimo de Kemeny, nessa subseção são apresentadas algumas.

Como já expressado exaustivamente nesse artigo, encontrar uma agregação ótima utilizando a regra de Kemeny(equacão 1) é NP-difícil, mesmo para quatro listas, a prova é a partir da redução do problema de *feedback arc set*[6]. Todavia, se utilizarmos como função de distância a Spearman footrule(equacao 3) é possível computar a agregação ótima de Spearman footrule em tempo polinomial, através da modelagem do problema como *minimum cost perfect matching* de um grafo bipartido. Assim, o problema de agregação é transformado em um grafo bipartido não direcionado $G = (C \cup P, A)$, onde cada candidato é convertido em um vértice $c \in C$ e as possíveis posições dos candidatos em vértices $p \in P$, e para todo par de c e p há arestas $(c,p) \in A$ e o peso da aresta (c,p) é dado por:

$$w(c,p) = \sum_{i=1}^{n} |\sigma_i(c) - p| \tag{6}$$

onde $\sigma_i(c)$ é posição do candidato c na i-ésima lista.

Usando a desigualdade de Diaconis - Graham temos :

$$\sum_i K(\sigma,\tau_i) \leq \sum_i F(\sigma,\tau_i) \leq \sum_i F(\sigma*,\tau_i) \leq 2\sum_i K(\sigma_*,\tau_i) \tag{7}$$

onde σ é a agregação ótima de Footrule e $\sigma*$ é a agregação ótima de Kendall. Portanto, temos um algoritmo polinomial 2-aproximado da ótima de Kemeny. Outro algoritmo 2-aproximado bem conhecido é: dado k listas, escolher uma aleatoriamente, ou então escolher a com menor distância kendall tau dentre elas [2], que vamos chamar de PICK-A-LIST.

Como a regra de Kemeny pode ser reduzida ao problema de *feedback arc set*, há várias abordagens na literatura que propõem algoritmos aproximados eficientes para resolução desse problema e, assim encontrar uma agregação próxima da ótima. Em [2] é apresentado um algoritmo 2-aproximado baseado no randomized quicksort chamado FAS_pivot que encontra uma solução aproximada para o problema em $O(n \log n)$. Também é apresentada uma junção entre o algoritmo PINK-A-LIST e o anterior, onde é mostrado que é um algoritmo $\frac{11}{7}$-aproximado. Esse algoritmo é uma das heurísticas apresentas como *baseline* desse projeto. Em [6] é apresentado um algoritmo baseado no problema de encontrar o caminho hamiltoniano em um grafo de torneio. Todo grafo de torneio tem ao menos um caminho hamiltoniano e ele tem uma relação intrínseca

com *problema feedback arc set*[3]. Utilizando a ideia apresentada em [3], existe um algoritmo baseado no paradigma de dividir e conquistar que encontra um caminho hamiltoniano em um torneio em $O(n \log n)$. Esse algoritmo será apresentado com mais riqueza de detalhes em seções futuras.

3 Baseline

Viu-se até o momento, que a tarefa de agregar todas as preferências individuais em uma única lista de preferência não é tão simples quanto parece ser.

Como discutido na Seção 1, uma abordagem bem natural é a regra de Kemeny, Equação (1). A lista ótima encontrada na otimização não necessariamente é única, tomando-se como instâncias de entrada as listas na Tabela 2.1, que tem um exemplo do paradoxo de Condorcet, percebê-se que todas as listas de preferência minimizam a Equação 1, logo, todas são ótimas.

As agregações ótimas de Kemeny possuem uma propriedade de voto majoritário, elas satifazem o critério de Condorcet [9]. Formalmente, seja σ ótima de Kemeny, se U e V particionam o conjunto de candidatos, e para cada $u \in U$ e $v \in V$ a maioria prefere u a v, então para todo $u \in U$ e $v \in V$ $\sigma(u) < \sigma(v)$.

Chama-se a conclusão deste teorema o critério de Condorcet generalizado (o critério de Condorcet pode ser informalmente definido como: se há um único elemento que bate todos os outros de uma maioria de votos, então ele deve ser o vencedor).

3.1 Baseline exato

O problema foi definido em termos de uma minimização sobre todos os *rankings* de n candidatos, do qual há $n!$ possíveis rankings. De fato, achar a ótima de Kemeny é NP-difícil. Assim, o *baseline* exato é um algoritmo força bruta que vasculha todo espaço de solução pela solução que minimiza a soma das distâncias Kendall tau. Obviamente, essa *baseline* será utilizada, somente, para instâncias pequenas do problema, pois sua complexidade $O(n!kn \log n)$ é impraticável.

Algorithm 1 $Optimal\_Kemeny\_Consensus(C, \{\tau_1, \tau_2, ..., \tau_k\})$

```
1: opt ← C
2: perm ← C
3: min ← Σ_{i=1}^{k} KT_dist(perm, τ_i) / k
4:
5: while next_permutation(perm) do
6:    dist ← Σ_{i=1}^{k} KT_dist(perm, τ_i) / k
7:    if min > dist then
8:       min ← dist
9:       opt ← perm
10:   end if
11: end while
12: return opt
```

Algorithm 2 $KT\_dist(\sigma, \tau)$

```
1: rank_τ ← get_rank(σ, τ)
2: (inversions, rank_τ) ← Sort_and_Count(rank_τ)
3: return 2(inversions) / (|rank_τ|(|rank_τ| − 1))
```

Teorema 3.1. *O algoritmo 2, KT_dist, é computado em* $O(|\tau| \log |\sigma| + |\tau| \log |\tau|)$. *Portanto,* $O(n \log n)$ *se* $n = \max\{|\tau|, |\sigma|\}$.

Demonstração. Na linha 1 a função *get_rank* retorna um arranjo que mapeia os elementos em τ com as posições em σ. Ou seja, seja $\sigma = \{x_1 < x_2 < x_3\}$ e $\tau = \{x_2 < x_1 < x_3\}$ tem-se $rank_\tau = \{2, 1, 3\}$, onde 2 é a posição do elemento x_2 na lista σ, 1 é a posição do elemento x_1

na lista σ. Se algum elemento em τ não existir em σ então é mapeado a uma posição igual a $|\sigma| + 1$. Se for utilizado na implementação da lista uma árvore rubro-negra onde os itens são as chaves e suas posições no rank o valor a ser recuperado, então executa-se *get_rank* em $O(|\tau| \log |\sigma|)$. A linha 2 pode ser executada em $O(|rank_\tau| \log |rank_\tau|)$[7], como $|rank_\tau| = |\tau|$ então o algoritmo *Sort_and_Count* é executado em $O(|\tau| \log |\tau|)$. Portanto, o tempo total do algoritmo é $O(|\tau| \log |\sigma| + |\tau| \log |\tau|)$. □

Teorema 3.2. *O algoritmo 1, Optimal_Kemeny_Consensus, encontra a permutação com menor distância kendall tau em* $O(n!k(m \log n + m \log m))$. *Onde* n *é o número de candidatos e* k *o número de listas e* $m = n$ *se forem listas completas, caso contrário* $m < n$. *Portanto, o algoritmo é* $O(n!kn \log n)$

Demonstração. Conjunto de candidatos C é dado pela união dos candidatos das listas $\{\tau_1, \tau_2, ..., \tau_k\}$ e $|C| = n$. E cada lista tem m candidatos, onde m pode ser $m = n$ se forem listas completas e $m < n$ caso contrário. As linhas 3 e 6 são executadas em $O(k(m \log n + m \log m))$, pois KT_dist é $O(|\tau| \log |\sigma| + |\tau| \log |\tau|)$ vide Teorema 3.1. O número de iterações do laço **while** da linha 5 é $O(n!)$, portanto a complexidade total do algoritmo é $O(n!k(m \log n + m \log m))$. □

3.2 Baseline aproximado

Em [6, 5] é apresentado o conceito de agregação ótima local de Kemeny.

Definição 3.2.1 (Ótima local de Kemeny)**.** Uma permutação π é uma agregação ótima local de Kemeny de listas parciais $\tau_1, \tau_2, ..., \tau_k$ se não há permutação π^* que possa ser obtida de π através de uma única troca de um par de elementos adjacentes e para qual $K(\pi^*, \tau_1, \tau_2, ..., \tau_k) < K(\pi, \tau_1, \tau_2, ..., \tau_k)$. Em outras palavras, é impossível reduzir a distância total dos τ's trocando um par adjacente [6].

Toda agregação ótima de kemeny é também localmente ótima, mas a recíproca não é verdadeira. Além disso, a agregação ótima local de Kemeny não é necessariamente uma boa aproximação. Todavia, é importante observar que a agregação ótima local de Kemeny satisfaz o critério generalizado de Condorcet.

Isso transforma o problema de uma busca global em uma busca local. Basicamente, inicia-se de qualquer ponto no espaço de soluções e procura-se localmente por trocas de pares adjacentes que alcancem o mínimo. A agregação ótima local de Kemeny pode ser encontrado em tempo polinomial. Uma implementação trivial seria rodar o *insertion sort* modificado. Claramente, o algoritmo teria complexidade $O(n^2)$. De fato, o problema de encontrar uma solução ótima local de Kemeny pode ser feita em $O(n \log n)$. Para isso precisamos definir o conjunto de listas parciais $\tau_1, \tau_2, ..., \tau_k$ como um digrafo majoritário T, também conhecido como torneio.

3.2.1 Modelagem em grafos

Definição 3.2.2 (Torneio/Digrafo majoritário)**.** Um torneio é um digrafo com todas as $\binom{n}{2}$ arestas presentes, cada com uma direção (das 2 possíveis). Seja $T = (V, A)$ um torneio então para todo par de vértices u e v existe uma aresta (u, v) ou (v, u).

Seja $T = (V, A)$ um torneio, para todo par de candidato $u \in V$ e $v \in V$ se a maioria prefere u a v então há uma aresta $(u, v) \in A$, caso contrário $(v, u) \in A$. Caso haja empate, existirá ambas arestas $(u, v) \in A$ e $(v, u) \in A$. Pode-se observar que nós relaxamos a definição de torneio, pois se o número de listas k for par pode haver empate, e portanto, tanto $\sigma(u) < \sigma(v)$ e $\sigma(u) > \sigma(v)$ são plausíveis. Todavia, se k for ímpar não há empate, pois sempre haverá maioria absoluta.

Exemplo 3.2.1. Seja $\tau_1 = \{A, C, B\}$, $\tau_2 = \{C, A, D\}$, $\tau_3 = \{C, D, B\}$, $\tau_4 = \{A, B, D\}$, $\tau_5 = \{C, B, D\}$ e $\tau_6 = \{A, C, B\}$ listas parciais de preferência sobre o conjunto de candidatos $S = \{A, B, C, D\}$.

Para tornarmos o exemplo 3.2.1 um torneio, todo candidato $a \in S$, $b \in S$ e $a \neq b$ contamos quantas vezes a é preferido a b e vice-versa. Desse modo produzimos a tabela 3.1, e a partir dela concluímos que $\tau(A) < \tau(B)$, $\tau(A) < \tau(D)$, $\tau(C) < \tau(B)$, $\tau(B) < \tau(D)$ e $\tau(C) < \tau(D)$ na maioria das vezes. E que também houve um empate na preferência entre A e C. Portanto, o

Par de candidadatos$_{ij}$	$\#\tau(i) < \tau(j)$	$\#\tau(i) > \tau(j)$
(A, B)	4	2
(A, C)	3	3
(A, D)	4	2
(B, C)	1	5
(B, D)	4	2
(C, D)	5	1

Tabela 3.1: Número de vitórias para cada par de candidato do Exemplo 3.2.1

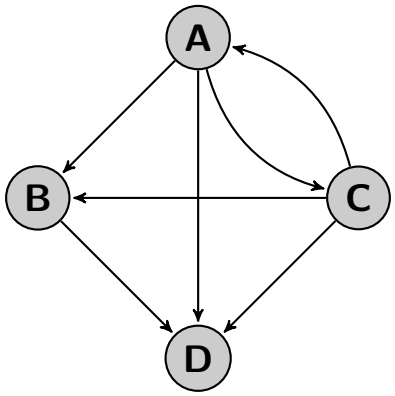

Figura 1: Grafo torneio do exemplo 3.2.1

torneio $T = (V, A)$ terá $V = S$ e $A = \{(A, B), (A, C), (C, A), (A, D), (C, B), (B, D), (C, D)\}$, a Figura 1 ilustra o grafo formado.

Agora o problema de encontrar a agregação local ótima de Kemeny é equivalente a encontrar o caminho hamiltoniono no torneio[6]. Em [3] é mostrado que qualquer algoritmo de ordenção baseado em comparação pode ser transformado em um algoritmo para encontrar caminho hamiltoniano em um torneio. Uma variação do mergesort será apresentado adiante como algoritmo para achar o caminho hamiltoniano.

O algoritmo acima produz uma ordem que é consistente com ponto inicial no seguinte ponto:

Se nós iniciarmos com uma lista π e então executarmos o algoritmo para chegarmos na lista π^*, então se $\pi^*(u) < \pi^*(v)$ nós devemos ter $\pi(u) < \pi(v)$ ou a maioria prefere u a v[6].

De fato para um dado ponto inicial existe uma única solução ótima que seja consistente. A única solução ótima local de Kemeny é chamada de *Kemenyzação local* da lista.

Portanto, essa é a ideia chave por trás das propostas: Escolher uma lista inicial de acordo com alguma heurística (Algumas heurísticas serão apresentadas com mais detalhes no decorrer dessa seção e da próxima), e a partir desse ponto inicial calcula-se sua *Kemenyzacão local*. Isso garante que a agregação gerada satisfaz o critério generalizado de Condorcet, e portanto é um bom consenso entre as listas (parcias) de preferência [6, 5], mas leve em conta que as heurísticas escolhidas também são importantes para que escolhamos um ponto inicial próximo ao ótimo.

3.2.2 Caminho Hamiltoniano e Kemenyzação local

Seja T=(V,A) o torneio definido pelas listas $\tau_1, \tau_2, ..., \tau_k$ e seja π uma ordenação dos candidatos. Como mencionado anteriormente, o problema de calcular o Kemenyzação local se torna o problema de encontar um caminho hamiltoniano no torneio. Suponha que nosso conjunto V de vértices esteja na mesma ordem que π, assim, ao encontrarmos o caminho hamiltonian de T utilizando o algoritmo 3 teremos a kemenyzação local ótima de π. E portanto, uma ordenção π^* que é consistente ao critério de Condorcet.

O algoritmo 3 - **Find_Hamiltonian** , é baseado no mergesort que utiliza o paradigma de programção dividir para conquistar. O algoritmo é baseado na idéia apresentada em [3], que diz que ao dividirmos o torneio T recursivamente em subtorneios $T^{'}$'s (linhas 6-11) até que achar um caminho hamiltoniano em $T^{'}$ seja trivial, por exemplo quando $|V^{'}| = 1$ que é o próprio

Algorithm 3 $Find\_Hamiltonian(T = (V, A))$

```
1: V_L ← ∅
2: V_R ← ∅
3: if |V| = 1 then
4:    return V
5: else
6:    V_L ← V[1..n/2]
7:    V_L ← V[n/2 + 1..n]
8:    Seja T_L = (V_L, A_L) o torneio induzido por V_L
9:    Seja T_R = (V_R, A_R) o torneio induzido por V_R
10:   (V_L) ← Find_Hamiltonian(T_L)
11:   (V_R) ← Find_Hamiltonian(T_R)
12:   return Merge_Hamiltonian_Path(T, V_L, V_R)
13: end if
```

V (linhas 3-4). Portanto, podemos mesclar os subcaminhos hamiltonianos dos subtorneios e formar o caminho hamiltoniano de T (linha 12).

Algorithm 4 $Merge\_Hamiltonian\_Path(T = (V, A), V_L, V_R)$

```
1: V ← ∅
2: l ← pointer_first(V_L)
3: r ← pointer_first(V_R)
4: while l ≠ end(V_L) and r ≠ end(V_R) do
5:    if (l, r) ∈ A then
6:       adicione elemento apontado por l em V
7:       avance com ponteiro l
8:    else
9:       if (r, l) ∈ A then
10:         adicione elemento apontado por r em V
11:         avance com ponteiro r
12:      end if
13:   end if
14: end while
15: Assim que l ou r chegarem ao final de suas respectivas listas, anexar os elementos restantes
    da outra lista em V
16: return V
```

A mesclagem dos subcaminhos é feito pelo algoritmo 4 **(Merge_Hamiltonian_Path)**, chamado na linha 12 do algoritmo Find_Hamiltonian.

Teorema 3.3. *Algoritmo 3, Find_Hamiltonian, pode ser computado em $O(n \log n)$, utilizando matriz de adjacência como representação do grafo.*

Demonstração. A complexidade do algoritmo 4 depende da representação de grafo usada. Se usarmos lista de adjacência onde a estrutura de dados utilizada para implementar a lista é uma árvore rugro-negra podemos executar a linha 5 e 9 em $O(\log n)$ tornando o algoritmo $O(n \log n)$. Mas, se for usado matriz de adjacência essas operações tem custo $O(1)$, e portanto, o algoritmo é $O(n)$. Dessa forma, podemos executar o algoritmo 3 em $O(n \log n)$. □

3.2.3 FAS_pivot

Algoritmo 2-aproximado proposto em [2], que reduz o problema de encontrar um permutação ótima de kemeny em um problema de *Minimal Feedback Arc Set*.

O algoritmo aproximado consiste em, também, definir as listas de preferências $\tau_1, \tau_2, ..., \tau_k$ como um torneio T, do mesmo modo exibido anteriormente. O algoritmo é definido a seguir: Seja $T = (V, A)$ um torneio definido pelas listas de preferências então retornar a ordem π gerada por $FAS\_pivot(T)$, algoritmo 5.

Algorithm 5 $FAS\_pivot(T = (V, A))$

```
1: V_L ← ∅
2: V_R ← ∅
3: if |V| = 1 then
4:   return V
5: else
6:   Escolha aleatoriamente um pivô i ∈ V
7:   for all vértice j ∈ V − {i} do
8:     if (j, i) ∈ A then
9:       adicione j a V_L
10:    else
11:      if (i, j) ∈ A then
12:        adicione j a V_R
13:      end if
14:    end if
15:  end for
16:  Seja T_L = (V_L, A_L) o torneio induzido por V_L
17:  Seja T_R = (V_R, A_R) o torneio induzido por V_R
18:  return FAS_pivot(T_L), i, FAS_pivot(T_R)
19: end if
```

O algoritmo tem uma propriedade desejável que, para toda entrada, ele requer $O(n \log n)$ comparações (uma média sobre todas as escolhas de pivô), se as comparações das linhas 8 e 11 tiverem tiverem custo $O(1)$.

Teorema 3.4. *Algoritmo 5, FAS_pivot, pode ser computado em $O(n \log n)$, utilizando matriz de adjacência como representação do grafo.*

Demonstração. Cada execução do FAS_pivot corresponde a uma árvore binária de pesquisa da seguinte forma: o pivô inicial é o nó raiz, o pivô do conjunto da esquerda V_L é a raiz do sub-árvore da esquerda, o pivô do conjunto da direita V_R é a raíz da sub-árvore da direita, e assim por diante. O número de comparações da execução do algoritmo é equivalente ao número de comparações durante a construção da árvore por uma sequência de inserções. Assim, o custo médio de comparações da FAS_pivot é igual ao custo médio de construção da árvore binária de pesquisa quando é inserido $\{x_1, x_2, ..., x_n\}$ de forma aleatória.

Seja C o custo de criação de uma árvore binária de pesquisa. Nós temos $C = \sum_i \sum_{j<i} c_{i,j}$, onde $c_{i,j}$ é uma variável aleatória binária expressando se na inserção de x_i houve uma comparação com x_j.

Pela linearidade da esperança, $E[C] = \sum_i \sum_{j<i} Pr(C_{i,j})$. Observando que $\{x_1, x_2, ..., x_j, x_i\}$ é uma permutação aleatória, a probabilidade de x_i ser adjacente a x_j é exatamente $\frac{2}{j+1}$. Portanto, $E[C] = \sum_i \sum_{j<i} \frac{2}{j+1} = O(\sum_i \log i) = O(n \log n)$ □

4 Heurísticas propostas

Seja $\tau_1, \tau_2, ..., \tau_k$ uma instância de *Rank Aggregation Problem* para um conjunto V de candidatos rotulados como $\{1, 2, ..., n\}$. Define-se um torneio $T = (V, A)$ como exibido na subseção 3.2.1. Todas as heurísticas serão baseadas na modelagem do problema em grafos apresentada na subseção 3.2.1.

4.1 FHP_greedy

Dado um $T = (V, A)$ como descrito previamente, então retornar a ordem π gerada por $FHP\_greedy(T)$ (Algoritmo 6) e então aplicar o algoritmo para encontrar a Kemenyzação local de π e gerar a ordem π^* que satisfaz o critério de Condorcet.

Analisemos o algoritmo 6. A linha 1 inicializa o cojunto da solução S. Nas linhas 2-5 para cada vértice $v \in V$ é abribuído a cor branca e é procurado o vértice com menor grau de entrada. O vértice s com menor grau de entrada é pintado de cinza e colocado na fila.

Algorithm 6 $FHP\_greedy(T = (V, A))$

```
1: S ← ∅
2: for each v ∈ V do
3:    v.color = WHITE
4:    s ← min_indegree(s, v)
5: end for
6: s.color ← GRAY
7: Q.Enqueue(s)
8: while Q ≠ ∅ do
9:    u ← Q.Dequeue()
10:   S ← S ∪ {u}
11:   for each v ∈ Adj[u] do
12:      v.indegree ← v.indegree − 1
13:      if v.color = WHITE then
14:         s ← min_indegree(s, v)
15:      end if
16:   end for
17:   if s ≠ NIL then
18:      Q.Enqueue(s)
19:      s.color = GRAY
20:   end if
21:   u.color ← BLACK
22: end while
23: return S
```

O laço **while** das linhas 8-22 itera enquanto houver algum vértice na fila Q que são os vértices que ainda não foram visitados. Esse laço **while** mantém o seguinte invariante:

No teste da linha 8, a fila Q consiste do conjunto de vértices cinzas.

Demonstração. **Inicialização :** Antes de qualquer iteração do laço **while** a fila Q contém o vértice de menor grau de entrada s que possui a cor cinza. Portanto, o invariante está correto.

Manutenção : No corpo do laço **while** nas linhas 9-10 o vértice com menor grau de entrada u é retirado da fila e colocado na solução S. Nas linhas 11-16 é percorrido os vértices brancos v adjacentes a u para atualizar seus graus de entradas e encontar o vértice adjacente s com menor grau. Daí, se há vizinho s então é inserido na fila e pintado de cinza e u é pintado de preto. Antes de iniciar a próxima iteração temos $Q = \{s\}$. Portanto, o invariante é mantido.

Término : O laço **while** termina quando $Q = \emptyset$ quando não há vértices cinzas, só pretos. Logo, invariante é verdadeiro. □

Teorema 4.1. *Algoritmo 6, FHP_greedy, pode ser computado em* $O(|V| + |A|)$.

Demonstração. O tempo de execução das linhas 2-5 é $O(|V|)$. As operações de enfileirar e desenfileirar tem custo $O(1)$, e portanto, o tempo total gasto para operações de fila é $O(|V|)$. Como o algoritmo percorre a lista de adjacência para cada vértice apenas quando ele é desenfileirado, então é percorrido somente uma vez. Desde que a soma das listas de adjacêcnia é $\Theta(|A|)$, o tempo total gasto para percorrer as listas de adjacência é $O(|A|)$. Portanto, o tempo de execução total do algoritmo é $O(|V| + |A|)$. Como nossa entrada é um torneio com $|V| = n$ e como é uma definição relaxada de torneio então $|A| \geq \frac{n(n-1)}{2}$. Portanto, temos que o algorimo é $O(n^2)$. □

4.2 Mixed

Seja $\tau_1, \tau_2, ..., \tau_k$ uma instância de *Rank Aggregation Problem* para um conjunto V de candidatos rotulados $\{1, 2, ..., n\}$. Definimos um torneio $T = (V, A)$ como exibido na subseção 3.2.1. Executa-se os algoritmos $Pick\_A\_List(\tau_1, \tau_2, ..., \tau_k)$, $FAS\_pivot(T)$ e $FHP\_greedy(T)$. Seja $\pi_{Pick\_A\_List}$, π_{FAS_pivot} e π_{FHP_greedy} as agregações resultantes de cada algoritmo, execute a Kemenyzação local em cada uma das listas gerando as permutações $\pi^*_{Pick\_A\_List}$, $\pi^*_{FAS\_pivot}$ e $\pi^*_{FHP\_greedy}$, e então retorne a melhor permutação dentre elas.

Teorema 4.2. *A heurística, Mixed, pode ser computada em* $O(km^2 + k^2 n \log n + kn \log n + n^2 + n \log n)$. *Onde* n *é o número de candidatos,* k *o número de listas e* $m = n$ *se as listas forem completas, caso contrário* $m < n$. *Portanto, a heurística tem complexidade de tempo* $O(kn^2 + k^2 n \log n + kn \log n + n^2 + n \log n)$.

Demonstração. Para criar o torneio $T = (V, A)$ é necessário percorrer todas as listas e contar para cada par de candidatos na lista quantas vezes ele ficou a frente do outro. Como há k listas e m candidatos em cada lista então essa operação leva $O(km^2)$ para ser executada. O algoritmo *Pick_a_List* pode ser executado em tempo $O(1)$ se for randomizado ou em $O(k^2 n \log n)$ se for escolhido a lista com menor distância Kendall tau do conjunto de listas. Pois, deve-se calcular a distância entre todos os pares de listas e escolher a lista que tem a menor média de distância. Como há $O(k^2)$ pares e podemos computar as distâncias entre duas listas em $O(n \log n)$[7], isso leva $O(k^2 n \log n)$.

Como demonstrado anteriormente, os algoritmos $FAS\_pivot(T)$ e $FHP\_greedy(T)$ tem complexidade de tempo $O(n \log n)$ e $O(n^2)$ respectivamente. Após a execução desses algoritmos encontramos a Kemenyzação local das permutações resultantes $\Pi = \{\pi_{Pick\_A\_List}, \pi_{FAS\_pivot}, \pi_{FHP\_greedy}\}$. Pelo teorema 3.3 a kemenyzação local é $O(n \log n)$. Seja $\Pi^* = \{\pi^*_{Pick\_A\_List}, \pi^*_{FAS\_pivot}, \pi^*_{FHP\_greedy}\}$ as kemenyzações locais de cada algoritmo, então para encontrar a permutação que possui a menor distância Kendall tau em $\Pi \cup \Pi^*$ leva $O(kn \log n)$. Portanto, a complexidade total da heurística é $O(km^2 + k^2 n \log n + kn \log n + n^2 + n \log n)$ □

5 Resultados e Discussões

Para análise experimental, nós implementamos os algoritmos apresentados nas seções anteriores em C++ e executamos no sistema operacional Linux Ubuntu 14.04 em um PC TOSHIBA-Satellite equipado com processador Itel i7 2.1Ghz quad-core com 8GB de memória ram. As implementações dos algoritmos - OKC(Optimal_Kemeny_Consensus), Pink_A_List, kPink_A_List (Pink_A_List com kemenyzação local), FAS_pivot, kFAS_pivot (FAS_pivot com kemenyzação local), FHP_greedy, kFHP_greedy (FHP_greedy com kemenyzação local), Mixed - são sequênciais.

Para a coleta dos dados experimentais, executamos o algoritmo 30 vezes para cada tipo de instância de entrada. Cada instância de entrada é um conjunto de k listas de preferências, essas podendo ser completas (seja seu tamanho m igual ao número de candidatos n) ou não ($m < n$). O tipo/classe de entrada é definido por essas três variáveis (k, m, n). Poranto, para gerar os gráficos das Figuras 3, 4, 5 e 6 foram criadas 30 instâncias sintéticas para cada tipo de entrada definida pela tripla (k,n,m), onde $k = \{3, 13, 23, ..., 90\}$, $n = \{3, 4, 5, ..., 8\}$ e $m = n$(para as figuras 4 e 6 m é igual $n - 2$), ou seja, 54 tipos de entradas diferentes, e para cada tipo de entrada foram criadas 30 instâncias distintas, portanto, 1620 entradas.

A métrica apresentada na Figura 5 e 6 PAM (Porcentagem de acertos médio) é calculada da seguinte forma: Seja $C = c_1, c_2, ..., c_s$ o cojunto de classes de entrada parametrizadas por (k, n, m), e seus elementos um conjunto de N instâncias de entrada distintas. Para cada uma das 1620 entradas é encontrada um agregação para cada um dos 8 algoritmos mencionados anteriormente. Cada uma das agregações tem um K-score que é média das distâncias Kendall tau entre agregação e as listas da entrada. Seja $K_i(A)$ o valor do K-score da i-ésima entrada de uma classe $c \in C$ produzido por um algoritmo A. A porcentagem de acertos do algoritmo A para um conjunto de entradas é dado por:

$$PA(A) = \frac{1}{N} \sum_{i}^{N} \{|K_i(A) - K_i(OKC)| == 0\} \tag{8}$$

A métrica PAM (Porcentagem de acertos médio) é dado por:

$$PAM(A) = \frac{1}{|C|} \sum_{i}^{|C|} PM_i(A) \tag{9}$$

A métrica apresentada nas Figuras 3 e 4 FAM (Fator de aproximação médio) é dada por:

$$FAM(A) = \frac{1}{|C|} \sum_{i}^{|C|} FA_i(A) \tag{10}$$

n	Gerar Torneio	OKC	FAS-pivot	FHP-greedy	Mixed
4.0	2.5e-04±(9.5e-05)	5.5e-04±(2.0e-04)	1.4e-05±(5.4e-06)	1.6e-05±(6.5e-06)	4.2e-04±(2.0e-04)
5.0	1.6e-04±(6.8e-05)	2.7e-03±(1.0e-03)	1.1e-05±(4.8e-06)	8.5e-06±(3.5e-06)	3.7e-04±(1.9e-04)
6.0	1.4e-04±(8.3e-05)	9.5e-03±(1.0e-03)	8.1e-06±(7.0e-07)	6.7e-06±(4.6e-07)	2.5e-04±(2.8e-05)
7.0	1.0e-04±(1.5e-05)	6.9e-02±(6.4e-03)	9.3e-06±(1.0e-06)	9.0e-06±(2.8e-06)	2.8e-04±(3.2e-05)
8.0	1.2e-04±(9.8e-06)	6.1e-01±(5.7e-02)	9.6e-06±(1.0e-06)	1.5e-05±(2.8e-06)	3.1e-04±(3.7e-05)
9.0	1.4e-04±(1.3e-05)	6.2e+00±(2.9e-01)	1.4e-05±(3.2e-06)	1.0e-05±(4.0e-07)	3.6e-04±(1.7e-05)
10.0	1.7e-04±(2.4e-05)	7.0e+01±(2.6e+00)	1.4e-05±(3.5e-06)	1.3e-05±(3.8e-06)	4.1e-04±(2.0e-05)

Tabela 5.1: Tempo médio de execução em segundos

onde FA(A), fator de aproximação, é dado por:

$$FA(A) = \frac{1}{N}\sum_{i}^{N} K_i(A)/K_i(OKC) \tag{11}$$

A métrica FAM indica quão distante da solução ótima a solução do algoritmo A está.

A Tabela 5.1 e Fig. 2 apresentam um comparativo dos tempos médios de execução dos algoritmos. A Tabela 5.1 compara o tempo de execução do *baseline* exato (OKC) com as heurísticas para instâncias pequenas. A Fig. 2 compara o tempo de execução entre todos os algoritmos que usam modelagem em grafos (não é levado em consideração o tempo gasto para criação do grafo). Como pode ser analisado na Table 5.1 o tempo de execução do algoritmo OKC aumenta celeremente a medida que a entrada aumenta, como já era esperado pela análise de complexidade e pelo fato do problema ser difícil. Dentre as heurísticas, a heurística que mais leva tempo a ser executada é a Mixed, como era esperado pela análise de complexidade apresentada nas seções anteriores. E obviamente, isso é devido ao fato que ela utilizada todas as outras heurísticas, no entanto, como todas as heurísticas são independentes é possível que a paralelização do código aumente a performance do algoritmo consideravelmente.

A figura 2 mostra o tempo médio de execução para entradas maiores e comprova que os algoritmos seguem as complexidades apresentadas nas seções anteriores. Como pode ser visto, o algoritmo FAS-pivot é o mais rápido dentre os que utilizam a modelagem de grafos, como era esperado já que foi mostrado que o algoritmo é $O(n \log n)$. Logo após ele vem o kFAS-pivot que também é $O(n \log n)$, porém ele tem uma constante escondida pelo O que é ligeiramente maior que a do FAS-pivot, já que ele executa a *kemenyzação* local que é $O(n \log n)$. Claramente, o FHP-greedy e kFHP-greedy seriam mais custosos que os anteriores. Como mostrado na análise de complexidade, apesar de serem lineares na representação do grafo ($O(|V| + |E|)$), eles são $O(n^2)$ em função do número de candidatos, pois o grafo é um torneio e, portanto, possui no mínimo $n(n+1)/2$ arestas. Como todos fazem parte do Mixed, o algoritmo FHP-greedy é um limite inferior para ele, dessa forma ele tinha de ser, realmente, o mais custoso dentre os algoritmos.

Pode-se observar nas figuras 5 e 6 que os algoritmos tiveram melhor resultado para listas completas do que listas parciais, exceto o algoritmo Pick-A-List que obteve FAMs similares tanto para listas parciais quanto para completas. Essa qualidade inferior nos resultados para lista parciais pode ser por causa do modo que é feito a modelagem em grafo, pois ao construir o torneio assumi-se que os candidatos que não se encontram na lista parcial de preferência estejam na posição $|\sigma| + 1$ e isso acaba gerando empates. No caso de listas completas isso não ocorre já que os candidatos nas listas são exatamente os candidatos do universo. Como visto, quando há empates é possível que haja o paradoxo de Condorcet. Portanto, esses empates gerados podem acabar gerando o paradoxo sem mesmo existir. Nós notamos que quando há paradoxo de Condorcet os resultados podem se distanciar do ótimo, sobretudo quando utilizado a *kemenyzação* local. Dessa forma, para esse caso particular observa-se que o a modelagem não funciona tão bem. Todavia, os resultados não foram ruins já que o algoritmo proposto, Mixed, se mostrou uma boa aproximação do ótimo, em média, tanto para listas parciais quanto para completas.

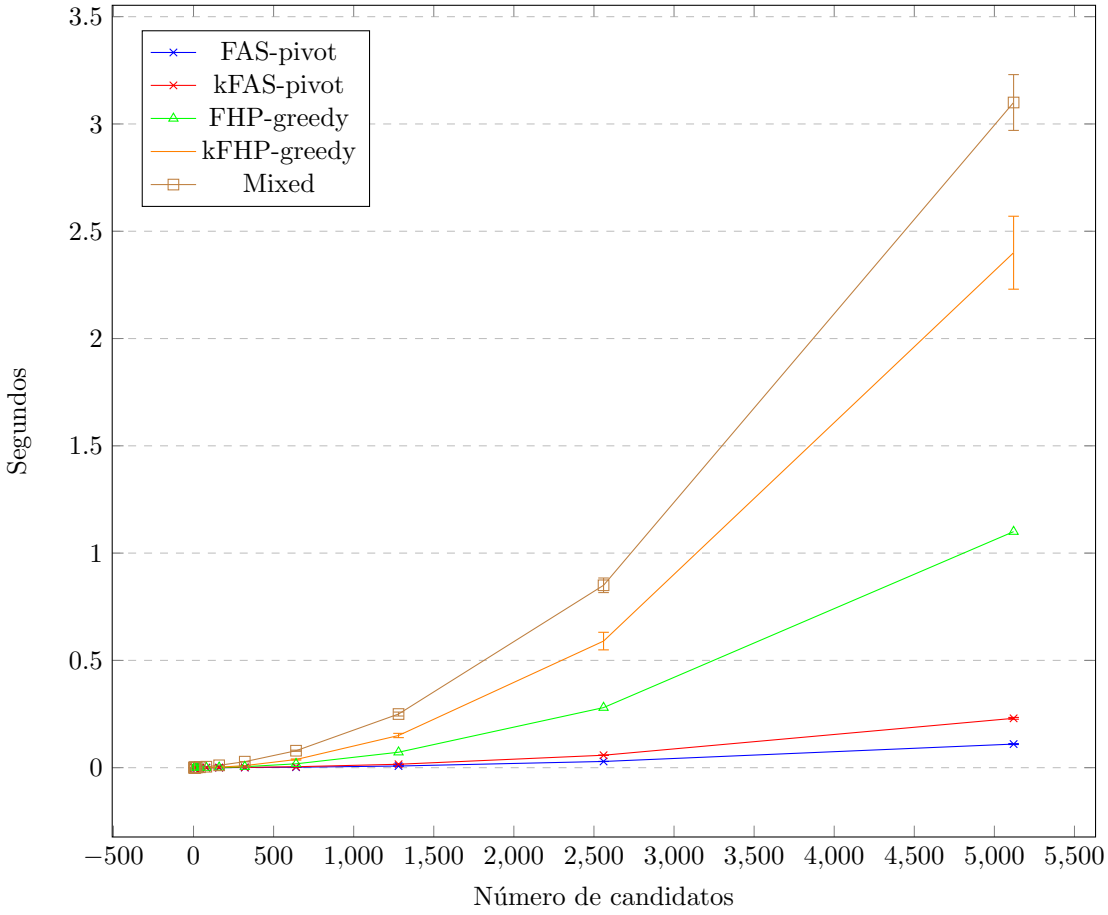

Figura 2: Tempo médio em segundos X Número de candidatos n (Número de listas k foi fixado como 10 e todas as listas são completas ($m = n$))

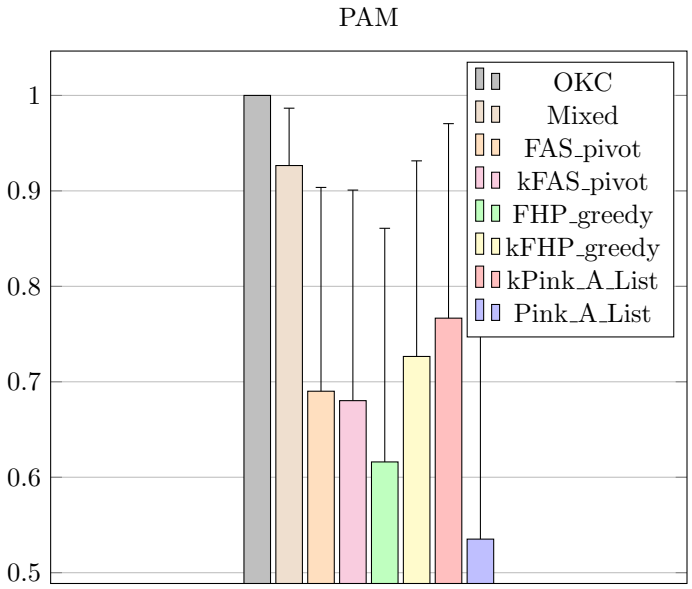

Figura 3: Porcentagem de acerto médio de cada algoritmo.

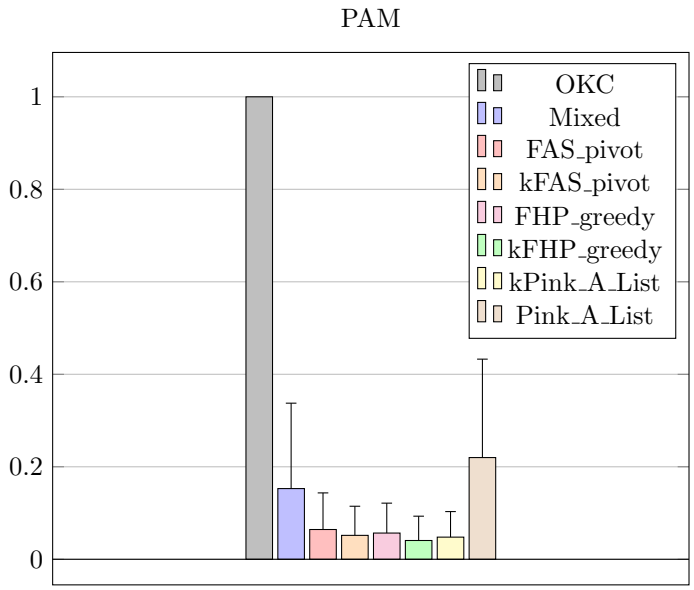

Figura 4: Porcentagem de acerto médio de cada algoritmo.

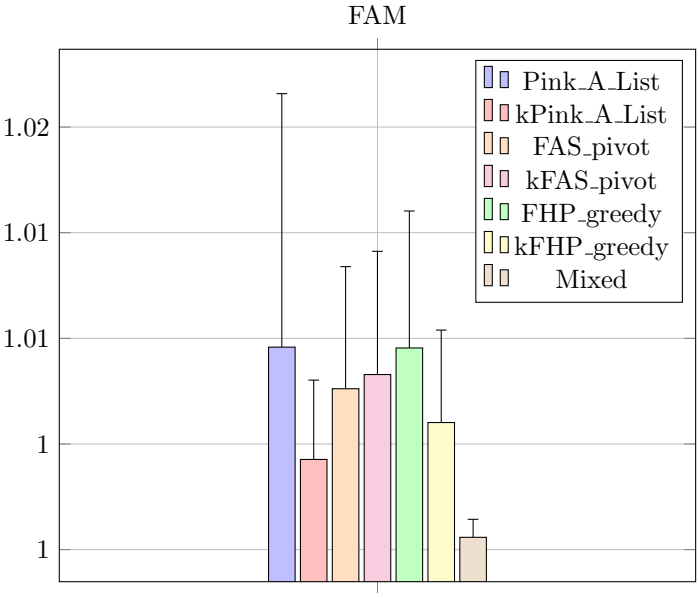

Figura 5: Média dos fatores de aproximação para cada algoritmo usando listas completas.

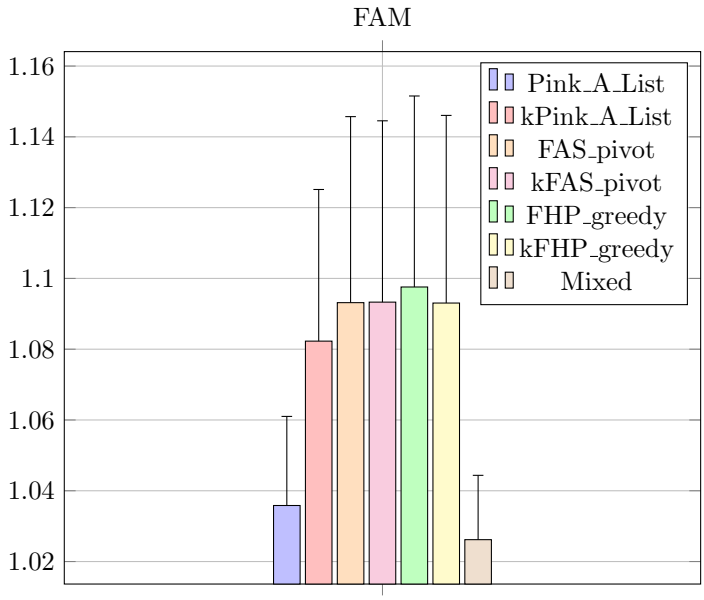

Figura 6: Média dos fatores de aproximação para cada algoritmo usando listas parciais.

6 Conclusão e Trabalhos Futuros

Esse trabalho propôs duas heurísticas para o *rank aggregation problem*, FHP_greedy e Mixed. Para validação dos algoritmos foram usados dados sintéticos de entrada e comparados com um *baseline* exato e um algoritmo aproximado.

Os resultados obtidos indicam a superioridade da combinação de diferentes heurísticas e *kemenyzação local*. Nossos resultados mostraram que o algoritmo Mixed teve melhores resultados que todos os outros algoritmos (exceto o ótimo) onde ele obteve 92% das respostas iguais a ótima para listas completas. Também foi visto que a *kemenyzação* local melhora a qualidade da solução quando não há paradoxo de Condorcet.

Como trabalhos futuros, nós pretendemos paralelizar a implementação da heurística Mixed e fazer uma análise mais teórica do porquê nossa heurísticas obteve resultados bons, mostrar e provar sua razão de aproximação. Além disso, nós notamos que a kemenyzação local aplicada a uma agregação π, gerada por algum algoritmo apresentado, produz uma ordem π^*. Se há o paradoxo de Condorcet para a dada entrada, então é possível a partir de π^* gerar outra permutação aplicando a *kemenynização* local em π^*, e assim, gerar π^{**}. Se for aplicado sucessivamente *kemenynizações* locais um momento será encontrado um ciclo. E nos testes feitos, foi notado que sempre uma dessas permutações, pertencente ao ciclo, é muito próximo da ótima solução. Uma proposta para trabalhos futuros seria explorar essa propriedade para desenvolver uma nova heurística.

Referências

[1] *Fixed-Parameter Algorithms for Computing Kemeny Scores – Theory and Practice.*

[2] N. Ailon, M. Charikar, and A. Newman. Aggregation inconsistent information: Ranking and clustering. 2005.

[3] A. Bar-Noy and J. Naor. Sorting, minimal feedback sets and hamilton paths in tournaments. 1988.

[4] J. J. Bartholdi, C. A. Tovey, and M. A. Trick. *Voting schemes for which it can be difficult to tell who won the election.* Social Choice and Welfare, 1989.

[5] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. *Rank Aggregation Methods for the Web.* WWW10, 2001.

[6] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. *Rank Aggregation Revisited.* 10th International World Wide Web Conference, May 2001.

[7] J. Kleinberg and E. Tardos. *Algorithm Design.* Addison Wesley, 2006.

[8] G. Lv. *An Analysis of Rank Aggregation Algorithms.* arxiv.org, 2014.

[9] H. P. Young. Condorcet's theory of voting. *AMERICAN POLITICAL SCIENCE REVIEW*, 84(4), December 1988.