Monografia de Conclusão de Curso

Aplicações de Programação Linear Inteira Mista à problemas de futebol

por:

Ricardo Augusto Trindade Carvalho

Bacharelado em Matemática Aplicada e Computacional

Instituto de Matemática e Estatística

Universidade de São Paulo

Orientador: Gabriel Haeser

25 de janeiro de 2015

Agradecimentos

Quero agradecer todos que participaram de alguma forma da minha passagem pelo IME: meus amigos, os professores, os funcionários do instituto, os meus colegas e aos amigos e treinadores da equipe de Atletismo, da qual fiz parte por vários anos.

Um agradecimento especial à professora Sônia que sempre foi muito solícita quando precisei de informações ao longo do curso. Às sempre atenciosas funcionárias da seção de alunos. Ao meu orientador, professor Gabriel Haeser, pela paciência em me orientar nesse trabalho. À minha família que sempre me apoiou em todos os momentos da minha vida.

Muito Obrigado.

Resumo

Nesse trabalho estudaremos dois problemas do universo esportivo que podem ser modelados por técnicas de Programação Linear Inteira. O problema do escalonamento de partidas que no qual formaremos uma tabela de jogos em rodadas num campeonato seguindo o sistema de ligas com n times. E o problema de eliminação para playoffs que visa determinar se um dos n times participantes tem a possibilidade ou a garantia de se classificar entre os m melhores times depois de uma rodad k.

Usaremos esses problemas para realizar um estudo introdutório do algoritmo Branch and Bound usado para resolver essa classe de problemas. Ao final resolveremos casos numéricos dos problemas descritos.

Lista de abreviaturas e siglas

RRT	Round Robin Tournament
t-RRT	t–Rounds RRT
HAP	Home–Away Pattern

Lista de símbolos

$S(k)$	k-ésimo elemento do conjunto ordenado S.
$\langle a, b \rangle$	Produto interno canônico entre a e b no R^n, também representado por $a'b$.
$\|S\|$	Cardinalidade do conjunto S.
$(x\ y)$	vetor de dimensão $m + n$, $x \in R^m$ e $y \in R^n$, também representado por $[x\ y]$.

Sumário

Introdução

Nesse trabalho estudaremos dois problemas de torneios de futebol que podem ser resolvidos com a solução de modelos de programação inteira.

O tipo de torneio que trataremos é chamado de *Round Bobin Tournament* ou simplesmente RRT. Nesse sistema o torneio contará com n times participantes. Para nos referir a esses n times usaremos uma enumeração arbitrária, representada pelo conjunto $T = \{1, \ldots, n\}$.

Um RRT é composto de 1 turno que conterá $n-1$ rodadas e cada rodada terá $\frac{n}{2}$ partidas. Em um turno cada time i joga uma única vez contra todos os outros contidos em $T \setminus \{i\}$ e em cada rodada cada time i jogará exatamente uma vez. Um RRT com t turnos será chamado de t-RRT.

Como as rodadas de um turno são ordenadas então naturalmente nos referiremos ao conjunto de rodadas de um RRT por $R = \{1, \ldots, n-1\}$.

Se n for um número ímpar então adicionaremos à T um time virtual v. Um jogo entre o time i e v serve apenas para marcar que i fica sem jogar na rodada em que essa partida estiver marcada. Dessa forma podemos tratar apenas os casos em que n é par.

Cada time estará associado a um estádio na cidade em que tiver sua sede. Quando um time i joga no estádio ao qual está associado dizemos que i é o mandante desse jogo. Chamaremos de visitante o time que não é o mandante. Toda partida entre dois times deverá ser disputada em um estádio no qual um dos dois será o mandante.

O vencedor de uma partida ganha 3 pontos e o perdedor 0 no caso em que um dos dois times ganhe o jogo ou ambos ganham 1 ponto no caso de empate.

Usaremos um RRT com as propriedades dadas acima para estudar modelos propostos para dois problemas, o problema do escalonamento de partidas e o modelo de eliminação para playoffs.

O primeiro modelo servirá para solucionar o problema de escalonamento de partidas num 1-RRT. Nesse problema queremos gerar um escalonamento de partidas dentro de rodadas que têm datas predefinidas. Em um 1-RRT com n times existirão $\binom{n}{n-2} = \frac{n}{2} \cdot (n-1)$ partidas para serem escalonadas.

Existem muitas formas de modelar e resolver esse problema e elas estão descritas nas seções 1 e 2 de Ribeiro et al. (2010). Nós consideraremos um 1-RRT e modelaremos o problema para resolvê-lo com métodos de programação inteira. Usaremos o modelo proposto na segunda seção de Briskorn e Drexl (2009) para essa finalidade.

Para o segundo problema usaremos dois modelos para determinar a possibilidade ou a garantia que um time tenha de se classificar entre as m melhores posições ao final do campeonato. Esses modelos são propostos em Ribeiro e Urrutia (2008).

Os métodos usualmente usados pela mídia desportiva para averiguar a possibilidade ou garantia de classificação podem falhar. Um desses métodos é a "probabilidade de classificação" que é construída com a proporção entre pontos obtidos e os pontos disputados por um time. No início do torneio estabelece-se um número de pontos que será usado como limiar entre o último classificado e o primeiro time não classificado entre os m melhores. Geralmente esse limiar é estabelecido usando-se estatísticas de torneios anteriores.

Nesse método uma equipe estará classificada se tiver mais pontos que esse limiar. Não é difícil criar situações na qual um time esteja entre os m melhores faltando k rodadas até o fim do torneio e que ele tenha mais pontos que o limiar estabelecido, mas ainda assim esse time pode perder todas as partidas restantes e ficar fora das m melhores posições ao final do torneio. Os modelos estudados não são suscetíveis a esse tipo de situação.

A apresentação desses problemas e seus respectivos modelos será realizada no capítulo 1.

No capítulo 2 faremos um breve resumo de programação linear e apresentaremos o método Branch and Bound para problemas com variáveis inteiras. Esse capítulo é principalmente baseado em Maculan e Fampa (2006) e em Chen, Batson e Dang (2010).

Parte desse trabalho consiste em implementar os modelos descritos e para isso diversos softwares foram usados entre eles o interpretador CPython, a biblioteca CyLP, os solvers cbc, mosek e LPSolve.

Os softwares e alguns casos de uso serão apresentados no capítulo 3.

Uma conclusão apontará as direções que podem ser tomadas a partir do que foi feito e um julgamento é apresentado sobre as implementações que foram realizadas para esse trabalho no capítulo 4.

1 Apresentação dos problemas

1.1 Problema do escalonamento de partidas

O problema do escalonamento de partidas para um RRT com n times participantes consiste em determinar o local e a rodada k onde uma partida entre os times i e j será jogada.

Para resolver esse problema implementaremos o modelo proposto na segunda seção de Briskorn e Drexl (2009). Outras fontes de referência para esse problema foram Ribeiro et al. (2010), Ribeiro e Urrutia (2012), Ribeiro (2012), Goossens e Spieksma (2009), Durán et al. (2007) e Recalde, Torres e Vaca (2013).

1.1.1 O modelo de Briskorn e Drexl

Para o modelo de Briskorn e Drexl é necessário construir o seguinte conjunto de variáveis:

$$x_{ijk} = \begin{cases} 1 & \text{se o time i joga como mandante contra o time j na rodada k} \\ 0 & \text{caso contrário} \end{cases}$$

Cada uma dessas variáveis será associada a um custo c_{ijk}. Esses custos deverão ser fornecidos como entrada para o modelo junto com o número de times n.

A função objetivo desse modelo é simplesmente a soma de todas as variáveis multiplicadas pelos seus respectivos custos. As restrições apenas garantem que o escalonamento gerado obedecerá as propriedades que façam o torneio ser um RRT.

Dessa forma o modelo é dado por:

$$\begin{aligned}
&\min \quad \sum_{i \in T} \sum_{j \in T \setminus \{i\}} \sum_{k \in R} c_{ijk} \cdot x_{ijk} \\
&\text{s.a.} \\
&\sum_{k \in R} (x_{ijk} + x_{jik}) = 1 && \forall i, j \in T, i < j && (1) \\
&\sum_{j \in T \setminus \{i\}} (x_{ijk} + x_{jik}) = 1 && \forall i \in T, k \in R && (2) \\
&x_{ijk} \in \{0, 1\} && \forall i, j \in T, i \neq j, k \in R
\end{aligned}$$

O conjunto de restrições (1) garante que cada time i enfrentará um time j numa única rodada do 1-RRT. Tanto faz percorrer i, j com $i < j$ ou $j < i$ pois x_{ijk} e x_{jik} são

usadas conjuntamente na restrição. O conjunto de restrições (2) determina que cada time i jogue apenas uma vez em cada rodada.

O número total de variáveis do modelo será $2 \cdot (n-1) \cdot \binom{n}{n-2}$. O conjunto (1) gerará $\binom{n}{n-2}$ restrições e o conjunto (2) gerará $2 \cdot \binom{n}{n-2}$ restrições.

Os custos c_{ijk} podem representar uma série de fatores tais como o gasto para organizar o jogo ou o custo de viagem do time j até o estádio de i ou mesmo outros custos possíveis. Se o nosso problema fosse de maximização esses valores poderiam representar receita ou lucro gerado com cada jogo.

1.1.2 Restrições para aumentar o equilíbrio do torneio

A solução do modelo de Briskorn e Drexl nos dará um escalonamento para um RRT, mas sua solução pode gerar escalonamentos não balanceados uma vez que o modelo de Briskorn e Drexl é relativamente simples e apenas formaliza em suas restrições a estrutura básica de um RRT.

Por exemplo, seja c_i o vetor contendo todos os custos associados às variáveis x_{ijk} nas quais um time i atue como mandante. Se para esse time tivermos $max(c_i) < c_{jj^*k}, \forall j \in T \setminus \{i\}$ então i jogará todas as suas partidas como mandante. Esse é um caso extremo de escalonamento não balanceado e serve para ilustrar o que queremos evitar.

Queremos que os times atuem como mandantes e visitantes aproximadamente o mesmo número de vezes e que não haja um grande número grande de rodadas nas quais um time jogue seguidamente como mandante ou visitante.

Adotaremos um conjunto de restrições que chamaremos de padrões mandante–visitante ou HAPs, *Home–Away Patterns*, no modelo de Briskorn e Drexl para sanar essa deficiência. A ideia é adicionar esses padrões no modelo de modo que forcemos o torneio a ser o mais equilibrado possível.

Padrões também são usados para reduzir o tempo computacional de solução do problema ao limitar o comportamento de conjuntos de variáveis e também para fixar datas de partidas entre alguns adversários. Isso costuma ser feito principalmente para jogos que são considerados "clássicos".

Há inúmeras maneiras de construir HAPs. Informações sobre HAPs e seus métodos de construção podem ser encontradas na seção 5.2.1 de Briskorn (2008). Vamos implementar apenas um método para gerar HAPs no modelo.

1.1.2.1 HAP quase aleatória

Usaremos nessa HAP a propriedade de que no conjunto de partidas de cada rodada teremos $\frac{n}{2}$ mandantes e $\frac{n}{2}$ visitantes e então geraremos um padrão seguindo as seguintes

regras:

1. $\frac{n}{2}$ times serão selecionados para o conjunto de mandantes
2. $\frac{n}{2}$ times serão selecionados para o conjunto de visitantes
3. Nenhum time pode ser mandante ou visitante por mais do que 2 jogos seguidos

Dessa forma supondo um torneio com $n = 4$ times um padrão possível seria dado por:

Padrão mandante-visitante — (HAP)			
Time	**Rodada 1**	**Rodada 2**	**Rodada 3**
1	Visitante	Visitante	Mandante
2	Mandante	Mandante	Visitante
3	Mandante	Visitante	Visitante
4	Visitante	Mandante	Mandante

Para adicionar essas restrições ao modelo consideraremos que quando um time joga como visitante o seu estado será dado pelo número 0 e quando joga como mandante o seu estado será dado por 1.

Podemos então construir conjuntos com os HAPs usando esses estados. Num torneio com n times chamaríamos esses conjuntos de $H_1, \ldots, H_n$. Para os HAPs dados acima associamos aos times de T os conjuntos $H_1 = \{0, 0, 1\}$, $H_2 = \{1, 1, 0\}$, $H_3 = \{1, 0, 0\}$ e $H_4 = \{0, 1, 1\}$.

Adicionamos a restrição (6) ao modelo para que esse considere os HAPs:

$$
\begin{aligned}
&\min \quad \sum_{i \in T} \sum_{j \in T \setminus \{i\}} \sum_{k \in R} c_{ijk} \cdot x_{ijk} \\
&\text{s.a.} \\
&\sum_{k \in R} (x_{ijk} + x_{jik}) = 1 && \forall i, j \in T, i < j \\
&\sum_{j \in T \setminus \{i\}} (x_{ijk} + x_{jik}) = 1 && \forall i \in T \\
&x_{ijk} \in \{0, 1\} && \forall i, j \in T, i \neq j, k \in R \\
&\sum_{j \in T \setminus \{i\}} (x_{ijk}) = 0 \iff \overbrace{H_i(k) = 0}^{\text{k-ésimo elemento de } H_i \text{ é } 0} && \forall i \in T, k \in R \qquad (6)
\end{aligned}
$$

Poderíamos ainda adicionar novas regras de equilíbrio para o torneio caso fosse necessário como aumentar ou reduzir o intervalo da regra 3 ou particionar o conjunto T e fazer com que times de uma mesma partição não joguem mais que um determinado número de rodadas consecutivas com times na mesma partição a que ele pertença.

Um problema para esse algoritmo de geração de HAPs é que não checamos em nenhum momento se os HAPs produzidos são viáveis. Esse problema é relativamente raro para casos onde $n > 14$ mas é comum quando $n = 6$. Seria interessante continuar o trabalho estudando métodos de verificação de viabilidade de HAPs.

1.2 Problema de eliminação para playoffs

Dado RRT com n times participantes queremos saber, em um dado momento do torneio, se um time i terá a classificação nas m primeiras posições garantida ou possível ao final do torneio.

Para solucionar esse questionamento usaremos dois modelos propostos em Ribeiro e Urrutia (2008).

Antes de apresentar os modelos algumas considerações são necessárias.

p_k^r representará a quantidade de pontos que o time k terá depois da rodada r.

A cada momento do torneio g_{kj} representará o número de partidas restantes entre os times k e j. Ao final do torneio $g_{kj} = 0$. Note que no caso de um 1-RRT $g_{kj} = 0$ se o jogo entre k e j já foi realizada ou $g_{kj} = 1$ no caso em que o jogo acontecerá numa rodada futura. No início de um t-RRT $g_{kj} = t$.

A todo o momento de um RRT teremos 3-uplas de valores inteiros $A(k, j) = (p_1(k, j), p_2(k, j), p_3(k, j))$ de tal forma que $p_1(k, j) + p_2(k, j) + p_3(k, j) = g_{kj}$, ou seja, $A(k, j)$ será uma partição de g_{kj} de maneira que $p_1(k, j)$, $p_2(k, j)$ e $p_3(k, j)$ representarão respectivamente números de vitórias, empates e derrotas do time k contra o time j nos jogos restantes ente eles. Veja que $p_1(k, j) = p_3(j, k)$.

Usando as 3-uplas $A(k, j)$ e p_k^r então, depois da rodada r, o número total de pontos que o time k pode conquistar no fim do torneio será dado por:

$$t_k = p_k^r + 3 \cdot \sum_{k \neq j} p_1(k, j) + \sum_{k \neq j} p_2(k, j)$$

1.2.1 O problema da classificação garantida

Com esse problema queremos saber, após uma dada rodada r, se um time k estará com a classificação garantida entre os m melhores times ao final do torneio.

Para isso definimos a posição final de um time i na competição como:

$$\underbrace{P_k = |\{j : 1 \leq j \leq n, j \neq k, t_j \geq t_k\}| + 1}_{\text{Número de times com mais ou a mesma quantidade de pontos que } k \text{ acrescido de 1.}}$$

Usando t_k e P_k podemos concluir que um time k terá a classificação garantida entre os m melhores se encontrarmos o inteiro mínimo G^k tal que para todas as possibilidades de valores das 3-uplas $A(k,j)$ tivermos $t_k \geq G^k$, e assim $P_k \leq m$.

Se existir o número máximo de pontos G^k_{max} de tal forma que exista ao menos uma 3-upla $A(k,j)$ tal que $P_k > m$ então G^k_{max} será o número máximo de pontos tal que k não tenha a classificação entre os m melhores garantida. Mas com isso $G^k_{max}+1$ seria a quantidade suficiente de pontos para que k se classifique entre os m melhores. O modelo que construiremos tenta achar G^k_{max}.

Considerando um t-RRT construímos as variáveis:

$$x_{kj} = \begin{cases} t & \text{se o time k tiver t vitórias sobre o time j} \\ \dots & \\ 2 & \text{se o time k tiver duas vitórias sobre o time j} \\ 1 & \text{se o time k tiver uma vitória sobre o time j} \\ 0 & \text{caso contrário} \end{cases}$$

$$y_j = \begin{cases} 1 & \text{se } t_j \geq t_k \text{ (time k não está com mais pontos que time j)} \\ 0 & \text{caso contrário} \end{cases}$$

No caso específico de um 1-RRT $t = 1$ e x_{kj} será dado por apenas dois casos.

Pelas definição que demos acima segue que $x_{kj} = p_1(k,j)$ e $p_3(k,j) = x_{jk} = p_1(j,k)$. Logo entre as g_{kj} partidas que serão disputadas entre k e j o número das que resultarão em empate é $[g_{kj} - (x_{kj} + x_{jk})]$. Isso é especialmente útil pois não temos uma variável que guarde o estado da quantidade de empates entre k e j e usando essas propriedades podemos reescrever t_k dado na última seção como $t_k = p^r_k + 3 \cdot \sum\limits_{k \neq j} x_{kj} + \sum\limits_{k \neq j} [g_{kj} - (x_{kj} + x_{ki})]$.

Usando esses dados teremos que o modelo para um time k será dado por:

$$\begin{aligned}
G^k_{max} = \quad & \max \quad t_k \\
& \text{s.a.} \\
& x_{ij} + x_{ji} \leq g_{ij} && \forall 1 \leq i < j \leq n && (7) \\
& t_j = p_j + 3 \cdot \sum_{i \neq j} x_{ji} + \sum_{i \neq j} [g_{ij} - (x_{ij} + x_{ji})] && \forall 1 \leq j \leq n && (8) \\
& t_k - t_j \leq M \cdot (1 - y_j) && \forall 1 \leq j \leq n, j \neq k && (9) \\
& \sum_{k \neq j} y_j \geq m && && (10) \\
& x_{ij} \in \{0, 1, 2, \dots, t\} && \forall 1 \leq i \leq n, 1 \leq j \leq n, i \neq j && \\
& y_j \in \{0, 1\} && \forall 1 \leq j \leq n, j \neq k && \\
& t_j \geq 0 && \forall 1 \leq j \leq n &&
\end{aligned}$$

O conjunto de restrições (7) limita o número de vitórias que cada time poderá obter nas partidas restantes entre k e j. Como $g_{ij} = g_{ji}$ a restrição parte de valores onde $1 \leq i < j \leq n$, daria na mesma fazer $1 \leq j < i \leq n$. O conjunto de restrições (8) nos dá o número de pontos obtidos por cada time ao final do torneio.

M é o valor máximo da diferença de pontos entre dois times em valor absoluto. Esse valor ocorre se um time tiver $t \cdot (n-1)$ vitórias e o outro tiver $t \cdot (n-1)$ derrotas. Logo a diferença máxima de pontos entre eles é dada por $M = t \cdot (n-1) \cdot 3$, considerando um t-RRT.

Considerando o M definido acima, o conjunto de restrições (9) garante que se $t_j < t_k$ então $y_j = 0$ e isso significa que o time k está com mais pontos que o time j, $y_j = 1$ caso contrário. A restrição (10) conta o número de times em $T \setminus \{k\}$ tal que $t_j \geq t_k$.

A restrição (10) e o conjunto (9) trabalham para fazer o problema ser inviável caso k já esteja garantido entre os m melhores, caso contrário a solução será G^k_{max}.

O número total de variáveis do modelo será $2\binom{n}{n-2} + n + n - 1$, sendo $2\binom{n}{n-2}$ variáveis x_{ij}, n variáveis t_j e $n-1$ variáveis y_j. Os conjuntos de restrições (7), (8) (9) gerarão respectivamente $\binom{n}{n-2}$, n e $n-1$ restrições e ainda teremos a restrição (10) no modelo.

1.2.2 O problema da classificação possível

Com esse problema queremos saber, após uma dada rodada r, se um time k estará com a classificação garantida entre os m melhores times ao final do torneio.

A posição final de um time k problema da classificação possível será dada por:

$$\underbrace{P_k = |\{j : 1 \leq j \leq n, j \neq i, t_j > t_k\}| + 1}_{\text{Número de times com mais pontos que } k \text{ acrescido de 1.}}$$

Usando t_k e P_k podemos concluir que um time k terá a classificação possível entre os m melhores se encontrarmos o inteiro mínimo P^k tal que entre todas as possibilidades de valores das 3-uplas $A(k, j)$ seja possível encontrar ao menos uma onde $t_k = P^k$, e assim $P_k \leq m$.

Se existir o número mínimo de pontos P^k_{min} de tal forma que exista ao menos uma 3-upla $A(k, j)$ tal que $P_i \leq m$ e $t_k = P^k_{min}$ então P^k_{min} será o número mínimo de pontos de foma que k tenha uma possibilidade de classificação ignorando critérios de desempate. O modelo que construiremos tenta achar P^k_{min}.

Considerando um t-RRT construímos as variáveis:

$$x_{kj} = \begin{cases} t & \text{se o time k tiver t vitórias sobre o time j} \\ \dots & \\ 2 & \text{se o time k tiver duas vitórias sobre o time j} \\ 1 & \text{se o time k tiver uma vitória sobre o time j} \\ 0 & \text{caso contrário} \end{cases}$$

$$z_j = \begin{cases} 1 & \text{se } t_j > t_k \text{ (time j está com mais pontos que time k)} \\ 0 & \text{caso contrário} \end{cases}$$

No caso específico de um 1-RRT $t = 1$ e x_{kj} será dado por apenas dois casos.

Usando as mesmas propriedades das variáveis x_{kj} e de t_k que foram usadas no problema de classificação garantida, podemos escrever nosso modelo para um time k como:

$$\begin{aligned} P^k_{min} = \quad & \min \quad t_k & & \\ & \text{s.a.} & & \\ & x_{ij} + x_{ji} \leq g_{ij} & \forall 1 \leq i < j \leq n & (11) \\ & t_j = p_j + 3 \cdot \sum_{i \neq j} x_{ji} + \sum_{i \neq j} [g_{ij} - (x_{ij} + x_{ji})] & \forall 1 \leq j \leq n & (12) \\ & t_j - t_k \leq M \cdot z_j & \forall 1 \leq j \leq n, j \neq k & (13) \\ & \sum_{k \neq j} z_j \leq m - 1 & & (14) \\ & x_{ij} \in \{0, 1, 2, \dots, t\} & \forall 1 \leq i \leq n, 1 \leq j \leq n, i \neq j & \\ & z_j \in \{0, 1\} & \forall 1 \leq j \leq n, j \neq k & \\ & t_j \geq 0 & \forall 1 \leq j \leq n & \end{aligned}$$

Os conjuntos de restrições (11) e (12) são usadas respectivamente pelas mesma razões que os conjuntos de restrições (7) e (8).

No modelo do problema da classificação possível os conjuntos de restrições (13) são usados de forma análoga aos conjuntos de restrições (9) do problema da classificação garantida, se $t_j > t_k$ teremos que $z_j = 1$ e assim o time j estará à frente do time k na classificação. O M utilizado nesse para esse conjunto de restrições será o mesmo do problema anterior.

A restrição (14) conta o número de times em $T \setminus \{k\}$ tal que $t_j > t_k$ e não permite que existam mais que $m - 1$ deles. Caso contrário o problema será inviável e nesse caso k não terá mais possibilidade de classificação. Caso o problema não seja inviável a solução será P^k_{min}.

O número total de variáveis do modelo será $2\binom{n}{n-2} + n + n - 1$, sendo $2\binom{n}{n-2}$ variáveis x_{ij}, n variáveis t_j e $n - 1$ variáveis z_j. Os conjuntos de restrições (11), (12) (13)

gerarão respectivamente $\binom{n}{n-2}$, n e $n-1$ restrições e ainda teremos a restrição (14) no modelo.

2 Programação Linear Inteira Mista e o método Branch-and-Bound

2.1 Introdução

CONTINUAR REVISÃO DAQUI

Vemos que pelas formas das funções objetivo e das restrições que todos os modelos apresentados no capítulo anterior são lineares e que poderíamos resolver o problema com métodos de programação linear não fossem as variáveis limitadas num conjunto discreto, para resolvermos exemplos numéricos dos modelos dados acima devemos então usar técnicas que levem isso em conta.

Estudaremos um método para lidar com esse tipo de variável.

As referências usadas para a parte de programação linear foram Maculan e Fampa (2006), Luenberger e Ye (2008) e Nocedal e Wright (1999). Já para a parte de programação inteira foram Maculan e Fampa (2006), Chen, Batson e Dang (2010) e Williams (2009).

2.2 Revisão de Programação Linear

Um Problema de Programação Linear é um problema que consiste em minimizar ou maximizar uma função linear sujeita à uma série de restrições também lineares, ou seja, suponha uma função linear $f(x)$ com $x \in \mathbb{R}^n$, como $f(x)$ é uma função linear podemos reescrever a função como $f(x) = c'x$ com $c \in \mathbb{R}^n$ representando os coeficientes da função $f(x)$, um problema exemplo é dado abaixo:

$$
\begin{aligned}
&\min \quad c'x \\
&\text{s.a.} \\
&x_1 + 3x_2 - 2.5x_3 \leq 4 \\
&2x_1 - 5x_4 - 0.5x_5 + x_6 \leq 5 \\
&4x_1 + 3x_2 + 2x_3 + x_4 \leq 10 \\
&x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}
$$

Nesse caso $x \in \mathbb{R}^6$.

Caso o problema não esteja escrito dessa forma ele pode ser transformado num problema dessa forma via substituição de restrições e de variáveis ou por multiplicações das restrições e da função objetivo pelo escalar -1.

Perceba que o conjunto de restrições pode ser organizado na forma matricial, no caso acima $A = \begin{bmatrix} 1 & 3 & -2.5 & 0 & 0 & 0 \\ 2 & 0 & 0 & -5 & -0.5 & 1 \\ 4 & 3 & 2 & 1 & 0 & 0 \end{bmatrix}$, e com o lado direito faremos $b = \begin{bmatrix} 4 \\ 5 \\ 10 \end{bmatrix}$, assim podemos escrever o problema como:

$$\begin{aligned} &\min \quad c'x \\ &\text{s.a.} \\ &Ax \leq b \\ &x \geq 0 \end{aligned}$$

2.2.1 Propriedades do conjunto de restrições

O conjunto de restrições é definido pela seguintes propriedades.

Definição 1. *Dado um número finito de de vetores no R^n, x_1, x_2, ..., x_n, uma combinação convexa desses pontos é um ponto dado por $\sum_{i=1}^{n} \lambda_i x_i$ com $\sum_{i=1}^{n} \lambda_i = 1$ e $\lambda_i \geq 0 \, \forall \, i \in \{1, \ldots, n\}$.*

Definição 2. *Seja $P \subseteq R^n$, $p \in P$ é um vértice de P se não existem, $x, y \in P$, $\lambda \in [0, 1]$ tal que $p = \lambda x + (1 - \lambda)y$.*

Definição 3. *O casco convexo de um conjunto $X \subseteq R^n$ é o conjunto gerado pelas combinações convexas de todos os vértices de X.*

Definição 4. *Um conjunto $X \subseteq R^n$ é convexo se X é igual ao seu casco convexo.*

Definição 5. *Seja $a \in R^n, a \neq 0$ e $b \in R$ o conjunto S dado por $S = \{x | a'x \leq b\}$ é chamado de semiespaço.*

Definição 6. *Sejam $a_1, a_2, \ldots, a_m \in R^n, a \neq 0$ e $b_1, b_2, \ldots, b_m \in R$, um poliedro é um conjunto P que satisfaz:*

$$P = \{x | Ax <= b\}$$

onde

$$A = \begin{bmatrix} a_1 \\ a_2 \\ \ldots \\ a_m \end{bmatrix} \quad e \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \ldots \\ b_m \end{bmatrix}$$

Corolário 1. *Um poliedro é uma intersecção finita de semiespaços.*

Teorema 1. *Todo semiespaço é convexo.*

Demonstração. Sejam $a \in R^n, a \neq 0$, $b \in R$ e $S = \{x | a'x \leq b\}$ um semiespaço, sejam $x, y \in S$ e $\lambda \in [0, 1]$, assim:

$$\langle a, \lambda x + (1 - \lambda)y \rangle = \lambda \langle a, x \rangle + (1 - \lambda)\langle a, y \rangle = \lambda a'x + (1 - \lambda)a'y \leq \lambda b + (1 - \lambda)b = b$$

□

Teorema 2. *A intersecção de conjuntos convexos é convexa.*

Demonstração. Sejam $X_1, X_2, \ldots, X_m \in R^n$ e seja $C = \bigcap_{i=1}^{m} X_i$, $\forall x, y \in C \rightarrow x, y \in X_1, X_2, \ldots, X_n$ como cada X_i é convexo segue que $\exists \lambda \in [0, 1]$ tal que $\lambda x + (1 - \lambda)y \in C$ pois $\lambda x + (1 - \lambda)y \in X_1, X_2, \ldots, X_n$. □

Corolário 2. *Todo poliedro $P \subset R^n$ é convexo.*

2.2.2 Um método para solução de problemas de programação linear

Como vimos no capítulo anterior o conjunto de restrições de um problema de programação linear forma um poliedro, que é um conjunto convexo. A área desse poliedro onde $x \geq 0$ será chamada de região viável. Vamos estudar agora as propriedades dos pontos que podem ser pontos ótimos de um problema de programação linear.

Vamos transformar o problema da forma que vimos, ao final da seção 2.2 para a forma aumentada:

$$\begin{aligned} &\min \quad c'x \\ &\text{s.a.} \\ &Ax + Ix_I = b \\ &x \geq 0, x_I \geq 0 \end{aligned}$$

A ideia é adicionar uma variável de folga, representadas pelo vetor x_I, para cada restrição do poliedro de restrições, ou seja, se tivermos m restrições em A então $I \in R^{m \times m}$ e $x_I \in R^m$. Essa forma é equivalente à original e será usada no restante desse capítulo.

Usando essa forma vamos particionar o problema para uma forma mais conveniente, seja $B \in R^{m \times m}$, B é inversível, e $N \in R^{m \times n-m}$ de tal forma que $[A\ I]$ é equivalente à $[B\ N]$. Particionaremos também $c = [c_B\ c_N]$ e $x = [x_B\ x_N]$. Assim reescrevemos o problema como:

$$\begin{aligned} &\min \quad c'_B x_B + c'_N x_N \\ &\text{s.a.} \\ &Bx_B + Nx_N = b \\ &x_B \geq 0, x_N \geq 0 \end{aligned}$$

Dessa forma podemos fazer com que $Bx_B + Nx_N = b \rightarrow Bx_B = b - Nx_N \rightarrow x_B = B^{-1}b - B^{-1}Nx_N$. Faremos então $x_N = 0$ para obter $\bar{x}_B = B^{-1}b$, com isso apresentamos a seguinte propriedade de $\bar{x}_B$:

Definição 7. *$\bar{x}$ é uma solução básica de $Ax + Ix_I = b$ se $\bar{x} = [\bar{x}_B\ 0]$, as variáveis associadas às componentes de $\bar{x}_B$ são denominadas de básicas e as associadas com x_N são não básicas.*

Chamamos solução básica viável adjacente todas os $\bar{x}_B^*$ obtidos ao trocarmos uma coluna de B por uma coluna de N e calcularmos novamente $\bar{x}_B$.

Se $\bar{x}_B \geq 0$ então chamaremos $\bar{x}$ de solução básica viável. Perceba que com isso nós temos um algoritmo para gerar soluções do problema de programação linear, basta gerarmos uma lista de m componentes de x para criar a matriz B e verificar se ela é inversível, por exemplo checando se ela tem o posto completo, então basta inverter a matriz B e calcular $x_B = B^{-1}b$, se fizermos isso para todas as combinações possíveis então bastará achar aquela que minimiza o problema.

O problema desse algoritmo é que ele é extremamente ineficiente se $dim(x)$ for muito grande, outro problema é que ele depende de um algoritmo de inversão de matrizes, o que novamente para dimensões grandes tornará o algoritmo custoso.

Um algoritmo melhor para lidar com isso é o algoritmo simplex. A ideia do algoritmo simplex é partir de uma base inicial e ir fazendo operações de troca entre elementos da base e de fora da base, trocando um elemento por vez, de forma a fazer isso sucessivas vezes até que não exista mais uma operação de troca que ache um novo x_B que minimize ainda mais a função objetivo.

Podemos dividir o simplex em três etapas:

1. *Inicialização:* Achar uma solução básica viável, e sua base associada.
2. *Iteração:* Achar uma solução básica viável melhor e adjacente à solução atual.
3. *Teste de otimalidade:* Testar se a solução atual é a ótima, caso contrário voltar ao passo 2.

A inicialização consiste em fornecer uma base inicial viável para o problema, podemos usar o algoritmo descrito acima para achar uma solução básica viável ou podemos fazer o uso de um problema auxiliar para achar uma solução básica inicial:

$$\begin{aligned} &\min \quad \sum_{i=1}^{m} y_i \\ &\text{s.a.} \\ &Ax + Iy = b \\ &x \geq 0, y \geq 0 \end{aligned}$$

Se resolvermos esse problema e o $\sum_{i=1}^{m} y_i > 0$ teremos que o nosso problema original é inviável, se $\sum_{i=1}^{m} y_i = 0$ e as variáveis $y_1, \ldots, y_n$ forem não básicas então iniciaremos o simplex para o problema original com a base do valor ótimo desse problema.

Caso alguma entre as variáveis artificiais $y_1, \ldots, y_n$ forem básicas, suponha a i-ésima, então para todas as colunas j de A não associadas com uma variável básica faremos $B^{-1}A_j$, se o i-ésimo elemento dessa multiplicação for diferente de zero então trocamos a variável artificial pela variável correspondente à A_j na base.

Faz sentido testar se a base obtida no problema auxiliar produz a solução ótima, a base produz a solução ótima se $x_B = B^{-1}b \geq 0$ e $\bar{c} = c - c_B B^{-1}A \geq 0$, esse é o teste de otimalidade.

Se o valor não for ótimo realizamos a fase de iteração, que tem dois passos, o primeiro consiste em decidir qual variável entrará na base e o segundo em decidir qual variável sairá da base. Ao fazer isso o algoritmo "caminha"na direção de uma base adjacente.

Para decidir qual variável entrará na base simplesmente selecionamos uma onde $\bar{c_j} < 0$. Antes de decidirmos quem será trocado faremos $u = B^{-1}A_j$, se $u \leq 0$ nós teremos que o problema é ilimitado, caso exista pelo menos uma componente onde $u_i > 0$ para decidir qual sairá da base selecionamos a variável que cumpra a condição $\bar{\theta} = \underset{u_i>0}{min} \frac{x_{B_i}}{u_i}$.

Então substituímos a variável associada à x_{B_i} com a variável associada à $\bar{c_j}$, fazemos o teste de otimalidade e caso a nova base não nos dê o valor ótimo repetimos a iteração do método até acharmos o valor ótimo ou determinar que o problema é ilimitado.

O simplex será usado como base para o algoritmo Branch and Bound que estudaremos a seguir.

2.3 Programação Linear com variáveis inteiras

O problema de Programação Linear com variáveis inteiras é basicamente o mesmo que foi apresentado, apenas adicionamos restrições que limitam um conjunto de variáveis a serem inteiras, exemplo:

$$
\begin{aligned}
&\min \quad c'x \\
&\text{s.a.} \\
&Ax \leq b \\
&x \geq 0 \\
&x_1, x_2, x_3 \in \mathbb{Z}^+
\end{aligned}
$$

Suponha por exemplo que para o problema acima tenhamos que $x \in R^n, n \geq 3$. Também podemos ter outro tipo de problema:

$$\begin{aligned} &\min \quad c'x \\ &\text{s.a.} \\ &Ax \leq b \\ &x \geq 0 \\ &x \in \{0, 1\} \end{aligned}$$

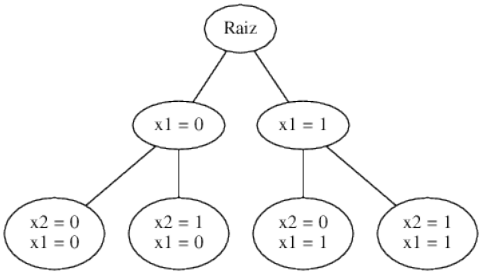
Nesse caso todas as componentes de x estão num conjunto limitado, e devem ser 0 ou 1.

Não podemos usar o simplex nesse tipo de problema uma vez que ele apenas considera variáveis continuas.

Uma estratégia possível para os casos onde as componentes de x deveriam estar num conjunto limitado seria fazer algo semelhante ao que apontamos na seção anterior, simplesmente enumerar todos os casos possíveis e testá-los individualmente para saber qual minimiza o nosso problema.

Suponha que no nosso último exemplo tenhamos que $c, x, b \in R^2$, esse é um caso extremamente simples onde essa estratégia nos daria a resposta, no total seriam apenas 4 casos possíveis, isso porém não é prático pois o crescimento de casos a se testar é exponencial, num problema com n variáveis binárias seria 2^n, para variáveis que possam assumir ainda mais que dois casos isso seria maior ainda.

Esse algoritmo é extremamente ineficiente mas ele nos traz uma das ideias que serão usadas para apresentar um algoritmo mais eficiente para esse tipo de problema. Ao enumerar todos os casos uma maneira lógica de organizar a ordem de teste seria se colocássemos todos os casos possíveis numa árvore, por exemplo, voltando ao caso do exemplo acima com variáveis binárias onde $c, x, b \in R^2$ teríamos:



A outra ideia que usaremos para construir um algoritmo é a ideia da relaxação, uma relaxação consistem em abandonar as restrições de integralidade. Por exemplo, para o caso do nosso segundo exemplo dessa seção teríamos que o problema relaxado seria dado por:

$$\begin{aligned} &\min \quad c'x \\ &\text{s.a.} \\ &Ax \leq b \\ &x \geq 0 \end{aligned}$$

Ou seja, seria simplesmente o problema sem a limitação de que as componentes de x deveriam ser inteiras.

O algoritmo Branch and Bound usa dessas duas ideias para resolver problemas com variáveis inteiras, a diferença é que a árvore será construída por partições no espaço de soluções. Essas partições são construídas dinamicamente a medida que resolvemos problemas de programação linear com relaxações dos subproblemas da árvore, chegamos à solução ótima justamente ao realizar esse processo.

Fazendo isso o algoritmo usa de uma estratégia de divisão dos problema inicial em problemas para o qual temos técnicas de resolução conhecida.

Definimos por N o conjunto dos nós de ativos. Inicialmente colocamos o problema relaxado no conjunto N, M_b é o valor do melhor problema inteiro, inicialmente definido por ∞.

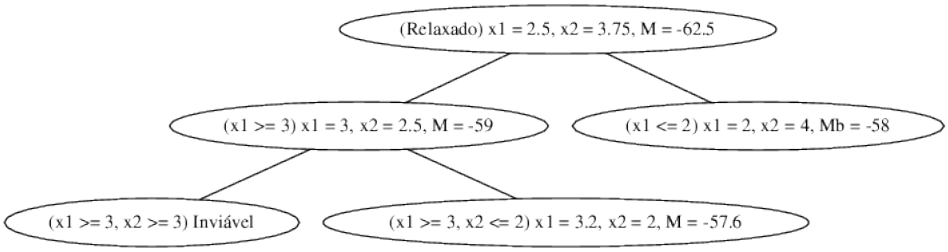
O Branch em Bound é dado pela repetição do pseudoalgoritmo abaixo:

1. Pegar um problema de N e resolvê-lo.
2. Se a solução M do problema relaxado obedece as restrições de integralidade armazenar ela em M_b, se $M < M_b$.
3. Encontrar o x_i mais inviável (mais distante de um número inteiro) cujo valor é dado por x_i^* e criaremos dois subproblemas filhos, adicionando ao primeiro a restrição $x_i \geq \lceil x_i^* \rceil$ e ao segundo adicionamos a restrição $x_i \leq \lfloor x_i^* \rfloor$ e adicionar esses problemas ao conjunto N.
4. Se um subproblema for inviável, a solução ótima de um subproblema obedecer as restrições de integralidade ou a solução ótima de um subproblema $M > M_b$ então tiramos aquele nó da árvore de subproblemas ativos e não resolveremos mais nós naquela subárvore.
5. Se N é vazio encerrar o algoritmo.
6. Voltar ao passo 1.

Para exemplificar o algoritmo consideramos o seguinte problema:

$$
\begin{aligned}
\min \quad & -13x_1 - 8x_2 \\
\text{s.a.} \\
& x_1 + 2x_2 \leq 10 \\
& 5x_1 + 2x_2 \leq 20 \\
& x \geq 0 \\
& x \in \mathbb{Z}^+
\end{aligned}
$$

A árvore para esse problema usando o algoritmo Branch and Bound será dada por:



E com isso encontramos nossa solução, que é dada por 58.

Além da clara vantagem desse algoritmo não necessitar que x esteja num conjunto limitado esse algoritmo se baseia no simplex, que já conhecemos, pra resolver problemas que originalmente o simplex não resolveria.

3 Resultados

Desenvolvemos então os modelos descritos no capítulo 1 e para fazermos a visualização desses resultados.

Todos os códigos aqui citados estão no repositório *https://github.com/rccrv/codigo-monografia.git*, os 2 mais importantes estão nos 3 apêndices abaixo. Todos eles foram rodados em ambiente Linux.

Os modelos diferentes estão divididos em arquivos diferentes, o arquivo Scheduling.py diz respeito ao modelo de escalonamento e o arquivo GeneralClassification.py diz respeito ao modelo de classificação para playoffs.

Para evitar perda de tempo com a preparação de dados de entrada (nomes dos times e custos) um script foi preparado para gerar a entrada para o modelo de escalonamento, esse script é o arquivo Team.py. Esse script gera uma lista de n times, $n \in \mathbb{N}$, par arbitrário, e também gera os custos usados no algoritmo de escalonamento.

Para fins de praticidade existem três modos de geração de custo, o primeiro gera um único valor que será usado como custo fixo para todos as partidas onde o time é mandante, o segundo gera uma lista de $n-1$ custos de onde é retirado uma amostra com reposição de $n-1$ custos que serão usados sequencialmente, ou seja, o primeiro custo corresponde a c_{i11}, o segundo a c_{i21} e assim por diante até $c_{i(n-1)\cdot(n-1)}$, o terceiro lê os custos de um arquivo de texto, e nesse caso os custos também serão usados sequencialmente, sendo necessário fornecer todos os $(n-1)^2$ em arquivos numerados, 1.txt, ..., n.txt. Esse script imprime o resultado na saída padrão sendo necessário redirecionar a saída para um arquivo.

Um exemplo de execução desse programa é dado por:

```
$ python Team.py 20 0 > teams.txt
```

O primeiro argumento corresponde ao número de times na competição e o segundo ao modo de geração custos usados, o argumento 0 corresponde ao modo 1, 1 ao modo 2 e 2 ao modo 3. A saída é redirecionada para o arquivo teams.txt.

Um exemplo de saída desse programa é dado por:

```
Time A;Guarulhos;SP;0;50
Time B;Campinas;SP;0;100
Time C;Rio de Janeiro;RJ;0;150
Time D;Salvador;BA;0;200
```

O arquivo está em formato *Comma-separated values* usando **;** como separador, o primeiro campo corresponde ao nome do time, o segundo corresponde à cidade onde o time

é mandante, o terceiro ao estado, o quarto ao tipo de geração de custos que foi usado e o quinto corresponde ao custo, caso o usuário tenha gerado ou fornecido custos sequenciais os custos corresponderão ao último campo e serão separados por vírgula.

Com essa saída pronta chamamos o programa Scheduling.py com o seguinte comando:

```
$ python Scheduling.py teams.py 1
```

O primeiro argumento corresponde ao arquivo de entrada do programa, gerado pelo script Team.py, o segundo aponta para o programa que queremos usar HAPs quase aleatórias, caso não desejemos usar HAPs usamos o comando:

```
$ python Scheduling.py teams.py
```

Esse programa usa a biblioteca CyLP pra preparar o modelo que será resolvido pelo cbc e gera como saída o arquivo schedule.txt, esse arquivo tem o seguinte formato:

```
1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0
0.0,0.0,0.0,0.0,0.0,1.0,0.0,1.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
```

Ele está em formato *Comma-separated values* usando **,** como separador, cada linha corresponde ao escalonamento de um time sendo simplesmente os valores obtidos para as variáveis x_{ijk}, nesse caso $n = 4$, por exemplo considere a primeira linha correspondente ao time 1, os três primeiros valores correspondem à $k = 1$, os três do meio correspondem à $k = 2$ e os três últimos para $k = 3$, para cada time em $T \setminus \{1\} = \{2, 3, 4\}$. No caso geral para a linha i cada r-ésima sequência de $n - 1$ valores corresponderá à situação da rodada $k = r$ do time i no conjunto $T \setminus \{i\}$.

O conjunto acima foi gerado sem obedecer à nenhum HAP, ao executarmos o modelo incluindo HAPs obtemos o arquivo de saída haps.txt contendo esses HAPS, esse arquivo segue o formato abaixo:

```
1;1;0
0;0;1
0;1;0
1;0;1
```

Novamente um arquivo em formato *Comma-separated values* usando **;** como separador, cada linha i corresponde à situação do time i e cada elemento representa uma rodada k, 1 representa que o time jogará como mandante e 0 como visitante, por exemplo o time 2 jogará fora de casa nas duas primeiras rodadas e será mandante na última. O escalonamento para esse HAP é dado por:

```
0.0,1.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0
0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0
0.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0
```

Com o arquivo schedule.txt de saída executamos o programa GeneralClassification.py da seguinte forma:

```
python GeneralClassification.py 2 > /dev/null
```

Primeiramente o programa lê o resultado do 1-RRT e faz o espelhamento para criar um 2-RRT, com isso feito o programa simula um torneio. A simulação foi feita usando a média de gols de times mandantes e visitantes do Campenato Inglês (*English Premier League*) 2014-15, o modelo divide essa média para os dois times, visitante e mandante por 90 minutos, calculando assim a esperança $\bar{X}_m$ e $\bar{X}_v$ de gols por minuto para o mandante e visitante respectivamente, com isso o número de gols é gerado usando uma distribuição binomial, ou seja imaginamos para as duas esperanças 90 testes de sucesso e fracasso que diz respeito à chance de fazer gols numa partida com as esperanças fornecidas acima.

A partir disso esse programa desenvolve o modelo de possibilidade para cada time i em cada rodada k, o programa escreve num formato que o LPSolve entenda e usa esse programa para traduzir o modelo para um arquivo de formato *Mathematical Programming System* (MPS), em seguida chama o solver mosek para resolver esse modelo, caso o time tenha possibilidades de classificação ele segue os mesmos passos para verificar se um time já está classificado ou não.

Por questões de performance e praticidade o algoritmo só resolve esses problemas para o segundo turno do 2-RRT, há sempre a possibilidade de um time ser eliminado no segundo turno independente de sua performance no primeiro e os problemas tem um tempo de execução muito alto se incluirmos o primeiro turno.

O único argumento do programa é o número m que será usado no modelo, o arquivo output é usado unicamente para que o mosek não escreva na tela.

Esse programa gera uma série de arquivos de saída contendo os pontos (points.r), a posição (position.r), a possibilidade (possibilities.r) ou garantia (guarantees.r) de que um time se classifique a cada rodada do 2-RRT.

Como achei que seria interessante ter uma representação gráfica da situação esses arquivos são gerados de forma que eles possam ser usados pela linguagem R para gerar um gráfico por rodada, isso não sacrifica a legibilidade de nenhum desses arquivos, por exemplo para o caso dos times acima, com um torneio simulado teríamos os seguintes abaixo, em todos os casos a lista $l[[i]]$ corresponde ao i-ésimo elemento do vetor *names*:

points.r

```
names <- c("Time D", "Time A", "Time C", "Time B")
l <- list()
l[[1]] <- c(0,1,2,5,8,9)
l[[2]] <- c(1,2,2,3,3,3)
l[[3]] <- c(3,6,9,9,9,12)
l[[4]] <- c(1,1,2,3,6,7)
m <- do.call(cbind, l)
```

position.r

```
names <- c("Time D", "Time A", "Time C", "Time B")
l <- list()
l[[1]] <- c(4,4,3,2,2,2)
l[[2]] <- c(2,2,4,3,4,4)
l[[3]] <- c(1,1,1,1,1,1)
l[[4]] <- c(3,3,2,4,3,3)
m <- do.call(cbind, l)
```

possibilities.r

```
names <- c("Time D", "Time A", "Time C", "Time B")
l <- list()
l[[1]] <- c(1,1,1,1,1,1)
l[[2]] <- c(1,1,1,1,0,0)
l[[3]] <- c(1,1,1,1,1,1)
l[[4]] <- c(1,1,1,1,1,0)
m <- do.call(cbind, l)
```

guarantees.r

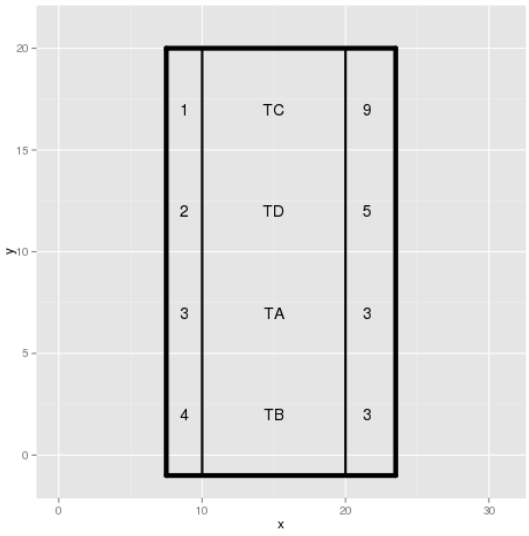
```
names <- c("Time D", "Time A", "Time C", "Time B")
l <- list()
l[[1]] <- c(0,0,0,0,0,1)
l[[2]] <- c(0,0,0,0,0,0)
l[[3]] <- c(0,0,0,0,0,1)
l[[4]] <- c(0,0,0,0,0,0)
m <- do.call(cbind, l)
```

Para o caso de possibilidade de classificação 1 no vetor significa possibilidade e 0 impossibilidade, já para o caso de garantia 1 significa garantia de classificação e 0 significa classificação não garantida.

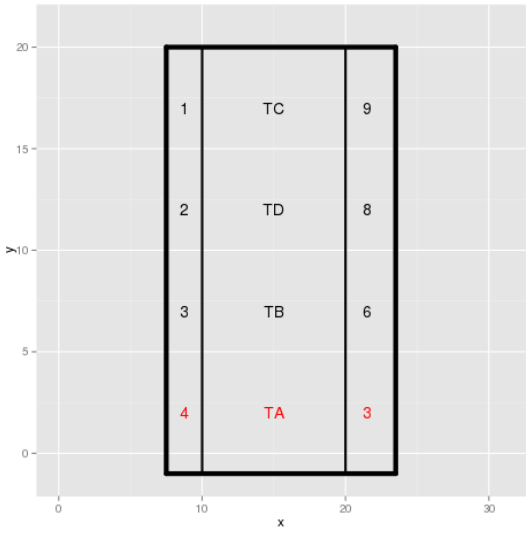
Com essas saídas geradas podemos chamar o script generate_graphics.r que gera um gráfico por rodada ordenando os times por posição e colocando suas siglas e seus

pontos nesses gráficos, se o time não tiver mais possibilidade de classificação ele aparecerá na cor vermelha na tabela, se ele tiver garantido aparecerá na cor azul.

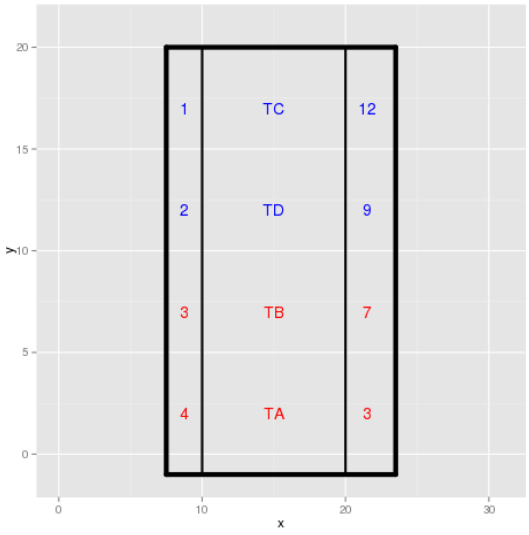
Para esse torneio temos os exemplos das tabelas abaixo, uma depois de cada rodada.



Rodada 4



Rodada 5



Rodada 6

Para apresentar todos os resultados dos dois modelos de uma forma só pode executar o script run-schedule-hap.py:

```
python run-schedule-hap.py
```

O script gera o arquivo saída.html que pode ser aberto em qualquer browser contém os HAPs (quando aplicável), o escalonamento e todas as tabelas que o generate_graphics.r gerar.

No caso do nosso problema exemplo com os 4 times dado acima ele gerou a seguinte saída:

HAPs (H: Time joga em casa, A: Time joga como visitante):

Rodada	**Time A**	**Time B**	**Time C**	**Time D**
1	H	A	A	H
2	H	A	H	A
3	A	H	A	H

Escalonamento (Time mandante x Time visitante):

Rodada 1:

Time A	x	Time C
Time D	x	Time B

Rodada 2:

Time A	x	Time B
Time C	x	Time D

Rodada 3:

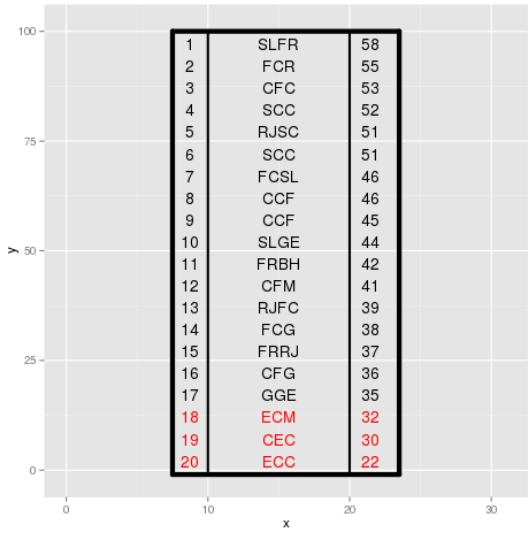
Time B	x	Time C
Time D	x	Time A

3.1 Casos interessantes

Alguns casos de análise são de especialmente interessantes, considere o caso abaixo com $n = 20$ times, são eles:

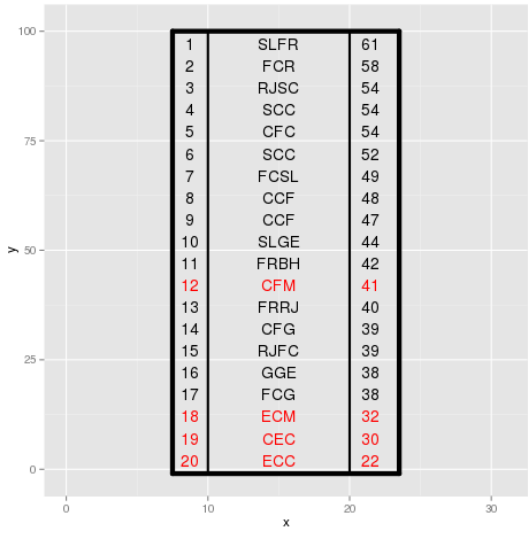
Clube de Futebol Manaus
Campinas Football Club
Football Club São Luís
Clube de Futebol Guarulhos
Futebol Clube Recife
Goiânia Grêmio Esportivo
São Luís Futebol e Regatas
Esporte Clube Campinas
Curitiba Clube de Futebol
Campinas Esporte Clube
Sport Club Curitiba
Sport Club Campinas
Futebol Clube Goiânia
Rio de Janeiro Football Club
Rio de Janeiro Sport Club
Futebol e Regatas Rio de Janeiro
Esporte Clube Manaus
São Luís Grêmio Esportivo
Futebol e Regatas Belo Horizonte
Campinas Clube de Futebol

A tabela para esse caso será impressa a sigla de cada time, por exemplo, o Esporte Clube Manaus como ECM e o Rio de Janeiro Football Club como RJFC e assim por diante. Observe essa tabela depois da 31ª rodada:

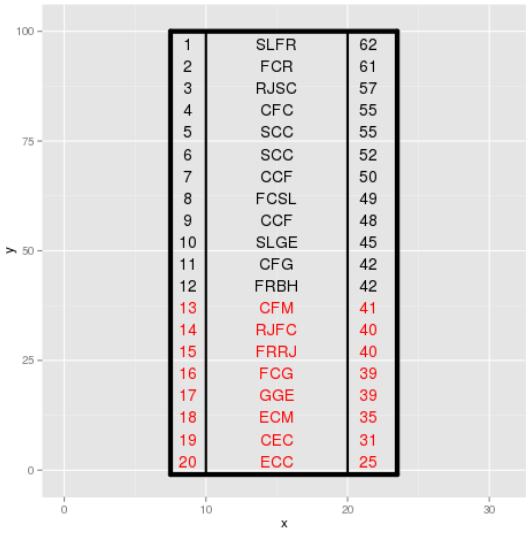


Rodada 31

A progressão da 32ª e 33ª rodada mostra um caso interessante:



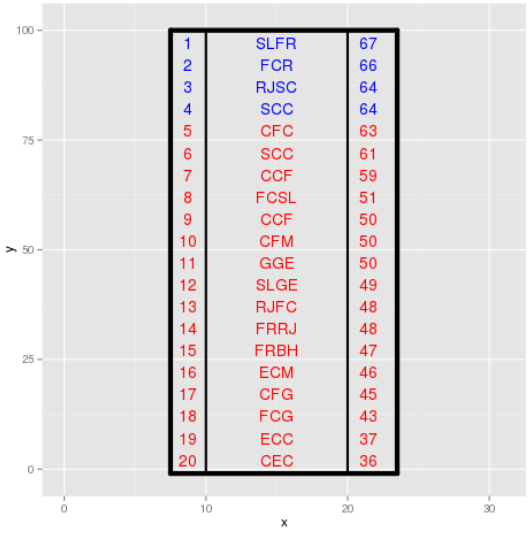
Rodada 32



Rodada 33

Nesse caso o time Clube de Futebol Manaus (CFM) está impossibilitado de se classificar antes do que outros times que ao final da 32ª estão com menos pontos que ele na tabela, é um caso interessante de aplicação do modelo pois isso significa que não há possibilidade de achar 3-uplas $A(i, j)$ de forma que ao menos uma garanta a possibilidade de classificação para essa equipe.

Ao final do torneio a tabela fica dessa forma:



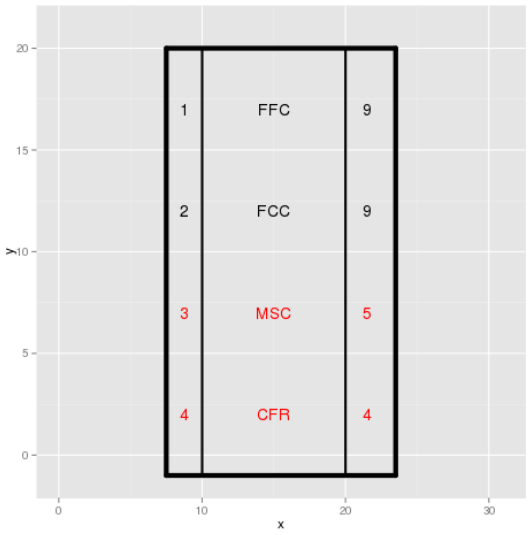
Rodada 38

Essa tabela também é interessante pois dois pares de times tem a mesma sigla, existem dois SCC e dois CCF na tabela, isso não permite diferenciar qual time termina o torneio em qual posição sem olhar os arquivos gerados depois da execução do GeneralClassification.py.

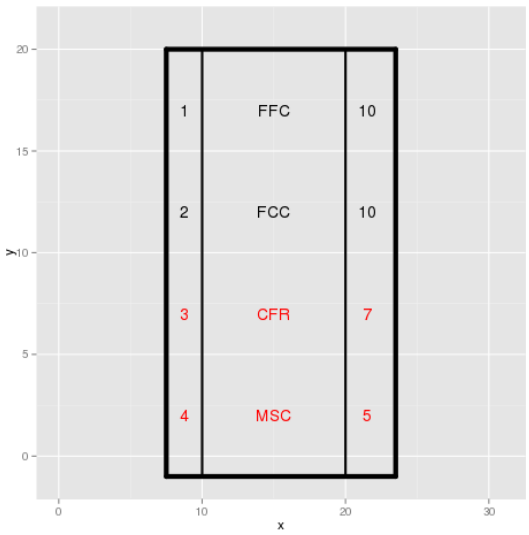
Outro caso interessante para $n = 4$, os 4 times são:

Futebol Clube Campinas
Curitiba Futebol e Regatas
Fortaleza Futebol Clube
Manaus Sport Club

Nesse caso $m = 1$ tivemos essa progressão das duas últimas rodadas:



Rodada 5



Rodada 6

O Campinas (FCC) e o Fortaleza (FFC) têm possibilidades de classificação e jogam entre si para decidir quem seria o primeiro, o resultado do jogo foi um empate de forma que ambos os times tenham o mesmo número de pontos ao final do campeonato, como o modelo de garantia não leva em conta critérios de desempate nenhum dos times aparece com classificação garantida, pois nesse caso um time pode estar com 10 pontos e estar eliminado. Já o modelo de possibilidade considera que como ambos tem pontos para

estar classificados, novamente ignorando critérios de desempate, então nenhum dos dois aparecerá como eliminado.

Se m for a última posição que garanta classificação e tivermos k times empatados com p pontos ao final do torneio então todos eles tem possibilidade de classificar e nenhum terá a garantia nesse modelo.

3.2 Benchmarks

Vamos analisar o tempo de execução dos algoritmos, para isso nós iremos fazer rodar os algoritmos com uma sequência de valores (n, m), onde n será o número de times que usaremos no Benchmark e m o número de melhores posições que passaremos pro algoritmo de classificação, faremos o problema para os conjuntos (4, 2), (8, 2), (12, 4), (16, 4) e (20, 4), vamos executar o algoritmo para cada modelo 5 vezes e extrair a média dos tempos de execução em segundos e o pior tempo entre todos, para o caso do modelo de escalonamento isso nos dará penas 5 valores, já para o problema de classificação nós teremos $5 \cdot (n-1)$ valores por execução, para o problema de classificação o programa que faz os Benchmarks é BenchmarkClassification.py.

A diferença do BenchmarkClassification.py para o GeneralClassification é que o primeiro faz os modelos de apenas 1 time e salva os tempos de execução dos dois algoritmos no arquivo benchmark-classification.txt, segundo executa os modelos para todos os times e não salva tempos de Benchmark.

Já para o caso do modelo de escalonamento, o programa Scheduling.py grava o arquivo benchmark-schedule.txt contendo o tempo de execução em segundos em todas as execuções.

Todos os testes foram feitos num computador com processador Intel Core i3-3110M e *kernel* Linux 3.16.0 compilado para a arquitetura x86-64.

Para o modelo de escalonamento, com as entradas de n que selecionamos acima obtivemos os seguintes resultados:

n	t **max (s)**	$\bar{t}$ **(s)**	t **max (s) com HAPs**	$\bar{t}$ **(s) com HAPs**
4	0.00137	0.00114	0.00137	0.00122
8	0.16289	0.14601	0.00867	0.00709
12	1.57919	1.55718	1.42005	1.26521
16	6.22243	5.88594	4.87649	4.33873
20	22.84706	22.50922	19.97932	18.53229

O modelo sem e com HAPs tem um tempo de execução bem curto, HAPs reduzem

o tempo de execução do algoritmo mas a redução é bem pequena em termos do tempo total de execução do algoritmo. A variância dos tempos também é baixa, em todos os casos a média está sempre muito próxima ao tempo máximo de execução.

Para o modelo de possibilidade de classificação, os resultados são apresentados na tabela abaixo:

n	t **max (s)**	$\bar{t}$ **(s)**
4	0.023	0.01693
8	0.42638	0.03407
12	0.59904	0.10588
16	5.83068	0.74849
20	174.96970	10.13815

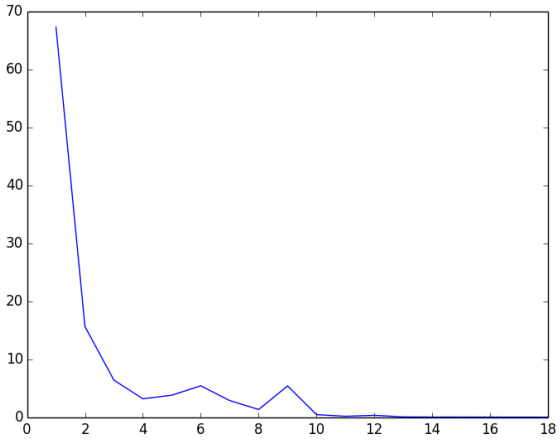
O modelo de possibilidades tem em média tempo de execução menor que o modelo de escalonamento, mas, com exceção do caso $n = 12$ e $n = 16$ tem um tempo de execução máximo pior que o modelo de escalonamento, junte-se a isso o fato de que geralmente esse modelo é rodado em todas as rodadas para todos os times e vemos que é nesse modelo que gastamos mais tempo.

Esse modelo tem uma variância bem alta principalmente graças as primeiras rodadas.

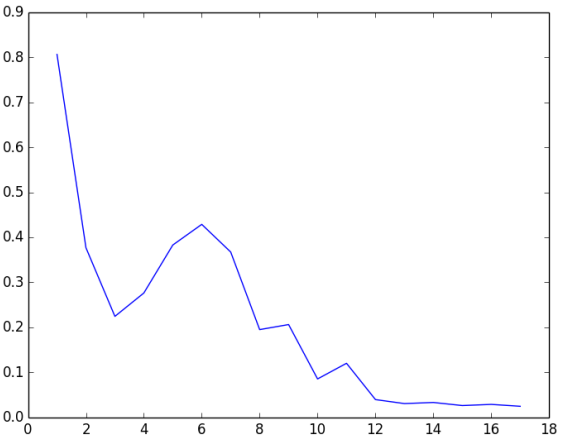
Uma observação interessante é que o modelo de modelo de garantia de classificação tem um comportamento bem diferente do de possibilidades:

n	t **max (s)**	$\bar{t}$ **(s)**
4	0.02255	0.01716
8	0.06319	0.02828
12	0.1896	0.06262
16	0.80679	0.10049
20	0.80628	0.10039

O tempo médio e o tempo máximo não crescem tanto em relação à n como no caso dos dois modelos anteriores, some-se a isso que esse modelo só é resolvido quando o modelo de possibilidades aponta que um time ainda tenha possibilidade de classificação. As duas imagens nos ajudam a visualizar a diferença de comportamento desses dois últimos modelos no caso onde $n = 20$ e $m = 4$.



Modelo de possibilidades



Modelo de garantias

Em ambos os casos o eixo das ordenadas representa o tempo em segundos e o eixo das abscissas as rodadas, sendo que a rodada 1 equivale à 21ª das rodadas totais. Pode-se ver claramente que todos os casos do problema da classificação possível com $x < 10$ têm um tempo de execução muito alto em relação ao problema da classificação garantida.

4 Conclusão

Vimos nesse trabalho duas aplicações interessantes de programação inteira com problemas relativamente grandes, esses modelos nos dão grandes possibilidades de testar Programação Linear Inteira no mundo real.

Existem novas direções para serem tomadas a partir daqui, o método de geração de HAPs usado é trivial, há uma seção de Briskorn (2008) que discute o conceito e permite possibilidades muito mais ricas, além disso várias das referências sobre esse assunto adotaram modelos mais complexos que seriam interessantes de serem explorados, existem ainda outros modelos interessantes que poderiam ser estudados nessa área de escalonamento esportivo.

Quanto ao modelo de possibilidades de classificação seria interessante explorar as razões da performance que apresenta, especialmente quando se considera os outros dois modelos. Também seria interessante estender os modelos para considerarem critérios de desempate, atualmente ignorados.

Uma outra possibilidade para esse modelo seria talvez a de verificar alguma forma de torná-lo um problema estocástico de forma que pudéssemos a qualquer momento avaliar a probabilidade de classificação de algum time de forma bem embasada na teoria de probabilidades.

A parte teórica permite um estudo muito mais aprofundado do que o que foi feito aqui, seja a possibilidade estudos de outros algoritmos como o Branch and Cut como estudar a teoria de programação inteira usando o ferramental de combinatória, o que aparece largamente na literatura quando se sai do estudo de algoritmos básicos como o Branch and Bound, o estudo de heurísticas para ajudar na resolução de problemas também seria interessante.

Para a parte de implementação creio que seria interessante melhorar o código e usar bibliotecas mais robustas, faltou tempo para poder explorar mais possibilidades de bibliotecas e solvers, um solver que me pareceu melhor é o scip, pois para os problemas apresentados ele foi tão ou mais rápido que o mosek na maioria dos casos e tem uma licença melhor para uso acadêmico.

Uma biblioteca que se mostrou especialmente promissora é a biblioteca pyomo, também para Python, ela permite que se possa modelar o problema através do Python (como a CyLP) mas dá a possibilidade de usar vários solvers, o glpk, o lpsolve, o mosek, o cbc, o scip, o gurobi, entre outros além disso com ela é possível distribuir a solução de problemas de programação inteira e pode ser que isso acelere razoavelmente a solução de

casos demorados, como alguns para o problema da classificação possível.

Uma outra possibilidade seria implementar o programa de forma que solvers diferentes pudessem tentar resolver o problema ao mesmo tempo, apesar de ter usado o mosek para o problema da classificação possível, pois no geral ele se mostrou mais rápido, existiram alguns casos nos quais o cbc e o glpk foram mais rápidos, seria interessante se nesse tipo de caso desse para usar alguns solvers ao mesmo tempo, de preferência num ambiente distribuído e pararmos todos os outros no momento que um deles encontrar uma resposta para o problema.

Além disso seria interessante transformar a implementação de ambos os problemas em bibliotecas independentes, mais bem escritas do a implementação atual, de forma que ela pudesse ser usada em outros ambientes, como uma aplicação web.

Referências

BRISKORN, D. *Sports Leagues Scheduling*. [S.l.]: Springer, 2008. ISBN 978-3-540-75517-3. Citado 2 vezes nas páginas 10 e 38.

BRISKORN, D.; DREXL, A. IP models for round robin tournaments. *Computers & Operations Research*, v. 36, n. 3, p. 837–852, 2009. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0305054807002298>. Citado 2 vezes nas páginas 7 e 9.

CHEN, D.-S.; BATSON, R. G.; DANG, Y. *Applied Integer Programming*. [S.l.]: John Wiley & Sons, 2010. ISBN 978-0-470-37306-4. Citado 2 vezes nas páginas 8 e 17.

DURÁN, G. et al. Scheduling the chilean soccer league by integer programming. *Interfaces*, v. 37, n. 6, p. 539 – 552, 2007. Disponível em: <http://pubsonline.informs.org/doi/abs/10.1287/inte.1070.0318?journalCode=inte>. Citado na página 9.

GOOSSENS, D.; SPIEKSMA, F. Scheduling the belgian soccer league. *Interfaces*, v. 39, n. 2, p. 109–118, 2009. Disponível em: <http://pubsonline.informs.org/doi/abs/10.1287/inte.1080.0402?journalCode=inte>. Citado na página 9.

LUENBERGER, D. G.; YE, Y. *Linear and Nonlinear Programming*. [S.l.]: Springer Science, 2008. ISBN 978-0-387-74502-2. Citado na página 17.

MACULAN, N.; FAMPA, M. H. C. *Otimização Linear*. [S.l.]: Editora UnB, 2006. ISBN 979-8-523-00927-3. Citado 2 vezes nas páginas 8 e 17.

NOCEDAL, J.; WRIGHT, S. J. *Numerical Optimization*. [S.l.]: Springer, 1999. ISBN 978-0-387-40065-5. Citado na página 17.

RECALDE, D.; TORRES, R.; VACA, P. Scheduling the professional ecuadorian football league by integer programming. *Computers & Operations Research*, v. 40, n. 10, p. 2478–2484, 2013. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0305054812002869>. Citado na página 9.

RIBEIRO, C. C. Sports scheduling: Problems and applications. *International Transactions in Operational Research*, v. 19, n. 1-2, p. 201–226, 2012. Disponível em: <http://www.dcc.ic.uff.br/~celso/artigos/sports-scheduling.pdf>. Citado na página 9.

RIBEIRO, C. C.; URRUTIA, S. A multi-agent framework to build integer programming applications to playoff elimination in sports tournaments. *International Transactions in Operational Research*, v. 15, n. 6, p. 739–753, 2008. Disponível em: <http://www2.ic.uff.br/~celso/artigos/futagent.ps>. Citado 2 vezes nas páginas 8 e 12.

RIBEIRO, C. C.; URRUTIA, S. Scheduling the brazilian soccer tournament: Solution approach and practice. *Interfaces*, v. 1, n. 42, p. 260–272, 2012. Disponível em: <http://www2.ic.uff.br/~celso/artigos/brsoccer8.pdf>. Citado na página 9.

RIBEIRO, C. C. et al. Scheduling in sports: An annotated bibliography. *Computers & Operations Research*, v. 1, n. 37, p. 1–19, 2010. Disponível em: <http:

//www2.ic.uff.br/~celso/artigos/Scheduling%20in%20sports%20C&OR.pdf>. Citado 2 vezes nas páginas 7 e 9.

WILLIAMS, H. P. *Logic and Integer Programming*. [S.l.]: Springer Science, 2009. ISBN 978-0-387-92279-9. Citado na página 17.