Fundação Getúlio Vargas - Escola de Matemática Aplicada
Disciplina: Programação Linear e Inteira
Prof. Luciano Monteiro

Solucionando o jogo *FlowFree* através de programação linear e inteira

Cristiana Aparecida Nogueira Couto

17 de abril de 2021

Resumo

Neste trabalho utiliza-se programação linear e inteira para resolução do jogo *FlowFree*, um tipo de quebra-cabeça lógico que consiste em conectar pontos de mesma cor num tabuleiro sem que as linhas se intersectem cobrindo todas as casas. Para tal, as restrições do problema foram codificadas para a utilização do pacote $GLPK$ da linguagem de programação *Julia*. Com o uso do solver, foi apresentado uma solução para um exemplo específico e comparado com uma resolução manual.

1 Introdução

Durante as aulas muitas aplicações de programação linear e inteira foram apresentadas, desde aplicações diretas em tomadas de decisão, organização, logística, além dos comuns problemas onde deseja-se maximizar lucro, minimizar custos, dentre outros. Diversas áreas do conhecimento se utilizam de programação linear para aplicação na resolução de diversos problemas. Há exemplos em economia, biologia, pesquisa operacional e muitos outros. Além disso, podemos utilizar a programação linear para a resolução de jogos. Alguns trabalhos, por exemplo, trazem os jogos sodoku[4] e o jogo da vida desenvolvido pelo matemático britânico John Conway[5] resolvidos com programação linear.

1.1 Quebra-cabeças lógicos

O *FlowFree* é uma variação de quebra-cabeças lógicos do tipo *Numberlink*. Esse tipo de jogo lógico - também conhecido como *Number Link, Nanbarinku, Arukone,* and *Flow* - foi popularizado por uma editora japonesa especializada em jogos, em particular, quebra-cabeças lógicos. A editora chamada *Nikoli* foi fundada em 1980. No entanto, há aparições desse tipo de problema datadas, por exemplo, de 1897 em uma coluna do matemático Sam Loyd [3].

O *Numberlink* consiste num jogo onde um grid $m \times n$ contém algumas casas com números e outras vazias. Os números aparecerem em pares e o objetivo é traçar linhas continuas que conectem os números iguais passando por todas as células do grid sem que essas linhas se cruzem. As linhas também não podem cruzar as células com números e devem passar no centro das células [2].

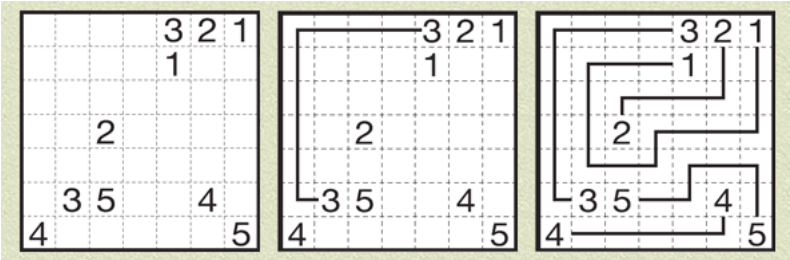

Figura 1: Mais à esquerda a imagem contém o grid inicial, ao seu lado as linhas vão sendo traçadas até a solução no grid mais à direita.
Fonte: [2]

1.2 O jogo *FlowFree*

O jogo *FlowFree* foi criado pelo estúdio *Big Duck Games*, uma empresa independente de jogos mobile fundado em 2012 na Flórida. O jogo já possui mais de 300 milhões de downloads e está disponível nas plataformas Android e IOS [1].

O jogo consiste num tabuleiro inicial $n \times n$ com pontos coloridos dispostos nas casas, cada cor com dois pontos iniciais. No aplicativo o usuário na tela inicial decide qual o tamanho do tabuleiro que pode variar entre $n = \{4, 5, 6, 7, ..., 15\}$. Em seguida, o usuário precisa conectar os pontos da mesma cor ao longo do tabuleiro sem que as linhas coloridas se cruzem. O desafio é resolvido quando todos os pontos estão conectados e todas as casas foram cobertas.

As imagens a seguir contêm a resolução para alguns exemplos de tabuleiros de tamanho $n = 5$, $n = 6$ e $n = 7$.

Figura 2: Resolução de exemplos com tabuleiros com $25, 36$ e 49 casas.
Fonte: reprodução *Big Duck Games LLC*

2 Programação Linear e Inteira

2.1 Representação matemática

Uma vez descrita as regras do jogo, precisamos convertê-las em variáveis e restrições lineares. A escolha de como se quer representar o problema matematicamente é crucial para que seja possível transformá-lo num problema de programação linear que seja equivalente ao jogo em questão.

Sendo assim, imagine o tabuleiro como um grafo cujos nós são as células, ou os pontos coloridos que estão no centro de cada casa. Cada nó tem no máximo quatro arestas que estão conectadas aos nós nas casas acima, à direita, à esquerda e abaixo. Note ainda que nós nas primeiras linha e colunas e últimas linha e coluna não possuem todas as quatro arestas. Os nós das casas do canto, por exemplo, só têm duas arestas.

Suponhamos que o jogo inicia com todas as arestas possíveis em cada nó, a solução do problema então consiste em apegar as arestas desnecessárias e atribuir exatamente uma cor as arestas remanescentes. Cada nó termina com no máximo duas arestas, uma vez que deve-se evitar intercessões entre as linhas, que são os caminhos que conectam os nós coloridos de início de uma mesma cor. Cada linha de uma cor é um caminho no grafo. Pode-se pensar também nesse problema como o seguinte: dado o nó inicial e o nó final e os pintando de k cores, queremos montar k caminhos que englobem todos os nós do grafo.

Feito isso, alguns modos de representar o grafo final obtido como uma solução viável para o problema pode ser, por exemplo, retornando uma matriz M de adjacência com elementos $m_{i,j} \in \{0, 1, 2, ..., k\}$, de modo que 0 indica a ausência da aresta e $m_{i,j} = c \neq 0$ que aquela aresta permanece e possui a cor atribuída ao número c. No entanto, ainda precisamos retornar também as cores de cada aresta. Além disso, cada nó só se conecta com no máximo outros quatro nós,

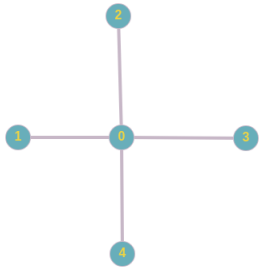

Figura 3: Arestas possíveis para o nó 0 em uma determinada célula do sub tabuleiro $(n-1) \times (n-1)$.
Fonte: autoria própria.

desse modo, estaríamos trabalhando desnecessariamente com uma matriz muito grande e esparsa.

2.2 Variáveis

Para definir as variáveis de decisão do problema, inicia-se pensando sobre quais escolhas que é preciso tomar para a resolução do problema. Deseja-se obter quais arestas serão mantidas e quais arestas descartadas. Além disso, deve-se definir quais as cores de cada casa no tabuleiro.

Definimos uma matriz N de adjacência limitada para as arestas que podem existir. Para cada aresta, há ainda um vetor de dimensão $1 \times c$, onde c é a quantidade de cores. Isso facilita as restrições para cores nas arestas. Para a restrição de ter exatamente uma cor em cada célula, uma matriz auxiliar será utilizada.

A seguir, as variáveis programadas para o solver:

```
@variable(model, 1 >= N[1:n*n, 1:4, 1:c] >= 0, Int)
@variable(model, 1 >= C[1:n*n, 1:c] >= 0, Int)
```

2.3 Restrições

- *Conectividade*: Cada nó só tem no máximo dois arestas saindo dele e no mínimo uma. Nós terminais só possuem uma aresta saindo dele;
- *Cores*: Todas as casas do tabuleiro possuem exatamente uma cor;
- *Cobertura*: As linhas cobrem todas as casas do tabuleiro.

- *Nós terminais*: São os nós que já iniciam com uma cor, de cada um deles só saí uma aresta;
- *Borda*: Os nós que estão na borda do tabuleiro não têm todas as quatro arestas que os nós que estão no meio do tabuleiro possuem;
- *Nós adjacentes*: Na representação matricial uma aresta aparece duas vezes, tanto como (u, v) quanto como (v, u). Na matriz elas precisam ter a mesma cor, pois representam a mesma aresta;
- *Arestas iguais verticais*: Como o que foi dito na restrição anterior, precisamos verificar se o vetor que representa as cores da aresta vertical que conecta dois nós é o mesmo.;
- *Cores das casas*: Cada casa só tem uma cor e não se pode conectar casas de cores diferentes;
- *Arestas inexistentes*: Nós onde o vetor de cores é nulo são inexistentes;

Cada casa do tabuleiro só possui uma cor (cada nó só tem uma cor). Se há uma aresta azul saindo do nó u, não pode haver uma aresta laranja chegando em u. No caso dos nós terminais não há esse problema porque só sai uma aresta dele. Além disso, a cor é fixa e já foi determinada nas restrições acima. Note que quando podem sair duas arestas de um nó, é possível acabar com duas arestas uma de cada cor, para isso que é adicionada a matriz auxiliar de cores do nó. Utiliza-se essa matriz para exatamente uma cor de cada casa no tabuleiro.

Todas essas questões pontuadas acima precisam estar embutidas nas *@contraints* que escrevemos para o problema. A depender do modo que elas são escritas é possível estar repetindo uma restrição mais de uma vez de forma diferente. Da mesma forma, colocar restrições de menos nos dá uma formulação equivocada do problema. O texto [6] faz um bom resumo das restrições necessárias e suficientes para a resolução do problema e serviu de inspiração para pensar neste trabalho em uma forma sucinta de estabelecer as restrições descritas na lista de modo que pudessem ser recebidas pelo $GLPK$ e que também obedecessem a restrição de linearidade. Outras formas, não lineares, de resolver o problema podem ser pensadas, mas esta além do escopo do presente trabalho.

A seguir os códigos com algumas restrições programadas para um caso genérico, onde o tabuleiro tem tamanho $n \times n$ e a quantidade de cores distintas utilizadas é dada pela quantidade c.

```
1
2       #Cada nó só tem no maxímo 2 arestas saindo dele e no mínimo 1
3       #Nós terminais só possuem uma aresta saindo dele
4       for i in 1:n*n
5           if i in nodesInput
6               @constraint(model, sum(N[i,1:4, 1:c]) == 1)
7           else
```

```
            @constraint(model, 1 <= sum(N[i,1:4, 1:c]) <= 2)
        end
    end

    #Restrições de cores
    for i in 1:n*n
        @constraint(model, sum(C[i, 1:c]) == 1)
        if i \notin nodesInput
            for j in 1:c
                @constraint(model, C[i,j] == sum(N[i,1:4,j])*0.5)
            end
        else
            for k in 1:c
                @constraint(model, C[i,k] == sum(N[i,1:4,k]))
            end
        end
    end

    #Cada aresta só possui uma cor e soma zero se ela não existe
    for i in 1:n*n
        for j in 1:4
           @constraint(model, sum(N[i,j, 1:c]) <= 1)
        end
    end

    #Cores do input nos nós terminais
    for j in 1:size(nodesInput)[1]
        @constraint(model, sum(N[nodesInput[j], i, nodesColors[j]]
    for i in 1:4) == 1)
    end


    #Os nós da borda não possuem algumas arestas 1= esquerda, 2= p/
     cima, 3= direita, 4 = p/ baixo
    #Começando pelos nós que não têm aresta esquerda
    for i in 0:n-1
        @constraint(model, sum(N[i*n + 1,1, 1:c]) == 0)
    end

    #Nós que não têm aresta p/ baixo
    for i in n*(n-1) + 1:n*n
        @constraint(model, sum(N[i, 4, 1:c]) == 0)
    end

    #Nós que não têm aresta p/ cima e nós que não têm aresta p/
    direita
    for i in 1:n
        @constraint(model, sum(N[i, 2, 1:n]) == 0)
        @constraint(model, sum(N[i*n, 3, 1:n]) == 0)
    end

    #A aresta (u,v) tem a mesma cor que a aresta (v,u)
    #Os vizinhos na horizontal e vertical que compartilham arestas
    precisam
    #ter as mesmas cores (começamos pelo subtabuleiro que possui
    todos os vizinhos)
    for i in n+2:n*n - (n+1)
```

```
        @constraint(model, N[i, 1, 1:c] .== N[i - 1, 3, 1:c])
        @constraint(model, N[i, 2, 1:c] .== N[i - 6, 4, 1:c])
        @constraint(model, N[i, 3, 1:c] .== N[i + 1, 1, 1:c])
        @constraint(model, N[i, 4, 1:c] .== N[i + 6, 2, 1:c])
    end
```

3 Resolução

A formulação do jogo em programação linear não possui uma função objetivo, busca-se aqui apenas uma solução viável que satisfaça as restrições. É testada a seguir a formulação com um tabuleiro em particular para verificar se a solução encontrada pelo solver, caso ele retorne uma solução, é compatível com a solução que podemos traçar manualmente.

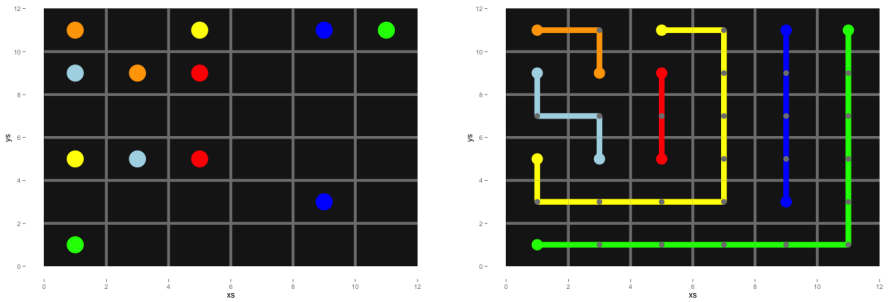

Figura 4: Um tabuleiro inicial 6×6 e à direita uma solução do jogo.
Fonte: O exemplo foi retirado de [6]

Apesar de um exemplo específico ser trazido, note que é possível testar qualquer tabuleiro com a função *flowFree*. As variáveis de entrada da função são o tamanho do tabuleiro, os nós que iniciam com cor listados da esquerda para a direita de cima para baixo, o número de cores e uma lista com os números que representam o cor de cada nó colorido inicialmente[1].

```
model = Model(GLPK.Optimizer)
nodesInput = [1, 3, 5, 6, 7, 8, 9, 19, 20, 21, 29, 31];
#Cores verde = 1, azul = 2, vermelho = 3, azul claro = 4, amarelo =
    5, laranja = 6
nodesColors = [6, 5, 2, 1, 4, 6, 3, 5, 4, 3, 2, 1];

function flowFree(model, n, nodesInput, c, nodesColors)
```

[1]Os códigos completos feitos em *Julia* se encontram no arquivo *flowfree.jl* que pode ser encontrado no repositório a seguir do *Github*: `https://github.com/Cristiananc/linear-programming/tree/main/FlowFree%20-%20Trabalho%201`

O número de variáveis aumenta consideravelmente com o aumento do tabuleiro. Nesse caso tem-se um total de $n^2(n * c + c) = 1080$ variáveis de decisão, pois $n = 6$ e $c = 6$. A seguir o plot obtido com a solução retornada pelo solver (as cores destacando a matriz foram feitas com uma caneta digital). Note que a solução obtida esta de fato correta.

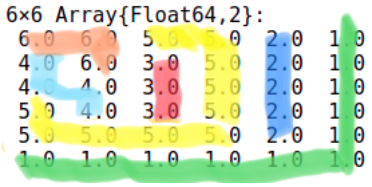

Figura 5: Solução encontrada pelo solver. Lembrando que verde = 1, azul = 2, vermelho = 3, azul claro = 4, amarelo = 5, laranja = 6.

4 Considerações finais

Com este trabalho foi possível verificar o uso da programação linear e inteira para resolver um quebra-cabeça lógico. Além disso, foi possível entender um pouco mais sobre a importância de modelar bem o problema, desde a formulação de restrições claras até a escolha de dimensões das matrizes de variáveis de decisão. Como futuras modificações complementares pretende-se implementar um algoritmo que possa gerar os tabuleiros iniciais, além de adicionar cores ao *plot* de solução.

Referências

[1] Big duck games. `https://www.bigduckgames.com/`. [Online; Acessado em: 17/04/2021].

[2] Numberlink. `http://www.nikoli.co.jp/en/puzzles/numberlink.html`. [Online; Acessado em: 17/04/2021].

[3] Adcock, A., Demaine, E., Demaine, M., O'Brien, M., Reidl, F., Sanchez Villaamil, F., and Sullivan, B. Zig-zag numberlink is np-complete. *Journal of Information Processing 23* (10 2014).

[4] Bartlett, A., Chartier, T., Langville, A., and Rankin, T. An integer programming model for the sudoku problem.

[5] Bosch, R. Integer programming and conway's game of life. *SIAM Rev. 41* (1999), 594–604.

[6] Narasimhan, R. Using r and integer programming to find solutions to flowfree game boards. https://www.r-bloggers.com/2013/07/using-r-and-integer-programming-to-find-solutions-to-flowfree-game-boards/. [Online; Acessado em: 13/04/2021].