Trabalho Computacional - Programação Não Linear
Picross e Ant Colony

Daniel Kiyoshi Hashimoto Vouzella de Andrade - 119025937

Novembro 2022

1 Problema: Picross

O problema escolhido foi a resolução de um puzzle de *picross*. Também conhecido por vários outros nomes (nonogram, griddler, hanjie, pic-a-pix, paint by numbers, ...), o puzzle tem como objetivo colorir uma grid $n \times m$ com quadrados, seguindo $n + m$ dicas (exatamente uma para cada linha e coluna). Por simplificação, só vamos considerar puzzles de tamanho $n \times n$.

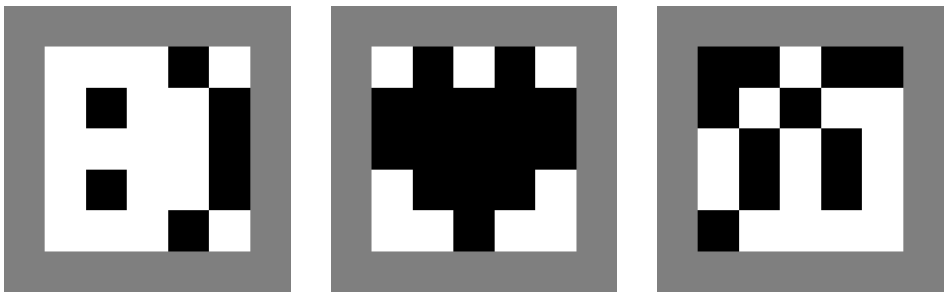
Cada dica é uma lista de números que indicam o tamanho dos blocos e sua ordem da sua respectiva linha ou coluna, e assim o tamanho da lista indica a quantidade de blocos dessa linha/coluna.

A figura 1 mostra dois exemplos de puzzles $n \times n$ já resolvidos, mas note que as dicas não estão incluídas e a borda cinza não faz parte do puzzle. As dicas são representadas por uma matriz $2 \times n$ de lista de números (eles estão "encaixotados"), de forma que a lista $(0, i)$ indica a ordem e tamanho dos blocos (esquerda para direita) da linha i (contando de cima para baixo). E, de forma similar, a lista $(1, i)$ indica a ordem e tamanho dos blocos (cima para baixo) da coluna i (contando da esquerda para direita). Em ambos os casos $0 \leq i < n$. Caso não tenha nenhum bloco na linha/coluna, aparece um 0 no lugar de uma lista vazia.

A seguir, estão as dicas dos puzzles 1a, 1b e 1c; a linha que está identada é o código em J e abaixo o resultado:

```
   NB. $ é usado para alterar as dimensões do array
   2 5 $ 1 ; 1 1 ; 1 ; 1 1 ; 1 ; 0 ; 1 1 ; 0 ; 1 1 ; 3
+-+---+-+---+-+
|1|1 1|1|1 1|1|
+-+---+-+---+-+
|0|1 1|0|1 1|3|
+-+---+-+---+-+
   2 5 $ 1 1 ; 5 ; 5 ; 3 ; 1 ; 2 ; 4 ; 4 ; 4 ; 2
+---+-+-+-+-+
|1 1|5|5|3|1|
+---+-+-+-+-+
|2  |4|4|4|2|
+---+-+-+-+-+
   2 5 $ 2 2 ; 1 1 ; 1 1 ; 1 1 ; 1 ; 2 1 ; 1 2 ; 1 ; 1 2 ; 1
+---+---+---+---+-+
|2 2|1 1|1 1|1 1|1|
+---+---+---+---+-+
|2 1|1 2|1  |1 2|1|
+---+---+---+---+-+
```

Ambos os problemas de existência e unicidade de uma solução (saber se existe uma solução e se ela é única) de um puzzle arbitrário de picross são *NP-completo*.



(a) Carinha Feliz de ASCII (b) Coração (c) Aleatório

Figura 1: Exemplos de picross 5x5

1.1 Formas conhecidas de resolução

- Depth First Search (Brute Force):

  Tenta pintar um pixel, se nenhuma dica quebrar tente de novo. Se alguma dica quebrar, troque a cor e continue. Se der errado com a nova cor, "volte" e tente de novo.

- Heuristic/Iterational:

  A ideia é aplicar uma sequência de regras simples para determinar a cor de algum pixel várias e várias vezes. A técnica é semelhante a um humano resolvendo o puzzle. Alguns puzzles mais complexos não podem ser resolvidos dessa forma, é comum usar um DFS quando uma iteração não encontra nenhum pixel novo.

- Integer Linear Programming:

  O puzzle é modelado como uma sequência de restrições lineares com uma função *dummy* de otimização.

2 Modelagem

O algoritmo escolhido foi o *Ant Colony*. O principal motivo para ele ter sido escolhido foi "formigas são legais!", na verdade, ele foi escolhido antes do problema em si. O problema é muito bom para essa heurística, pois é o problema é facilmente traduzido em escolher um caminho sobre um grafo e grafo é gerado bem simples. *Ant Colony* funciona com um simples loop, no qual a cada iteração são executados os passos:

1. Construir caminhos

   Nesse passo, são construídos *ants* caminhos e o melhor deles, de acordo com a *função objetivo*, é guardado. Cada caminho, deve ser escolhido aleatóriamente em função da *distribuição do feromônio* e da *heurística*. Geralmente são usados os parâmetros α e β para equilibrar esses valores que formam os pesos dos caminhos.

2. Atualizar o feromônio

   O melhor caminho da iteração c é usado para atualizar o feromônio τ_e em cada aresta e do grafo usando a seguinte fórmula:

   $$\tau_e \leftarrow (1-\rho)\ \tau_e + in(c,e)\ \frac{\rho\ \tau}{len(c)}$$

   onde ρ é a taxa de evaporação do feromônio (um parâmetro), τ é a soma de feromônio em todas as atestas (uma constante), $in(c,e)$ é uma função que retorna 1 se c passa pela aresta e e 0 caso contrário e $len(c)$ são por quantas arestas c passa.

3. Checar a condição de parada

   Existem duas condições mais comuns: checar se "acabou o gás", podendo ser o número de iterações ou tempo de execução; checar se achamos uma solução. Nem sempre é possível ou fácil usar a segunda, mas nesse caso é possível e relativamente fácil. Quando for decidido que se deve parar, retorna-se o melhor caminho encontrado até agora (não necessariamente dessa iteração).

2.1 Função Objetivo

A *função objetivo* deve impor uma ordenação sobre os caminhos, de forma que quanto mais próxima da solução "melhor" ela é.

2.2 Heurística

A *heurística* é uma forma de aumentar a chance de escolher um caminho mais provável de ser melhor. Também deixa pouco provável caminhos que aparentam ser ruins. Idealmente ela vai apontar para caminhos que minimizam/maximizam a *função objetivo*.

3 Implementação

A linguagem escolhida para a implementação foi `J`: uma linguagem orientada a arrays, da mesma família que `APL`. Não foi usada nenhuma biblioteca para construir o algoritmo, mas foi usada `viewmat` para fazer as imagens apresentadas nesse relatório.

3.1 Representação do Grafo e das Dicas

Para um puzzle $n \times m$, construímos um grafo com $1 + n \cdot m$ vértices numerados de $-1 \leq v < n \cdot m$ e $2 \cdot n \cdot m$ arestas nomeadas (p, b), com $0 \leq p < n \cdot m$ e $b \in \{0, 1\}$.

-1 é o vértice inicial do caminho, só está alí por formalidades; os outros $0 \leq v < n \cdot m$ representam um pixel de coordenadas $(x(v), y(v))$ do puzzle. A princípio, x e y podem ser quaisquer funções que, em conjunto, não repetem nem deixam sobrar nenhum pixel. Por simplicidade, vamos usar $x(v) := \text{mod}(v, m)$ e $y(v) := \left\lfloor \frac{v}{m} \right\rfloor$, mas, pela natureza da linguagem escolhida, isso vai "aparecer" como a ausência de uma função que reordena uma possível solução no código.

A aresta (p, b) sai do vértice $p - 1$ e chega no vértice p, b indica a cor do pixel p: se $b = 0$ é vazio/branco, é preto caso contrário. Na prática podemos "jogar os vértices fora" e só prestar atenção nas arestas, isso torna grafo é tão simples a ponto de não aparecer explicitamente na implementação.

A representação das dicas já foi descrita na primeira seção, mas não foi dito que ela só funciona para puzzles quadrados (todas as linhas de uma matriz devem ter o mesmo tamanho). Não é difícil criar uma outra representação que suporte puzzles retangulares, basta encaixotar cada lista, mas não tinha pensado em suportar puzzles retangulares durante a implementação. A dica do puzzle 1a ficaria assim:

```
   (1 ; 1 1 ; 1 ; 1 1 ; 1) ; < 0 ; 1 1 ; 0 ; 1 1 ; 3
+--------------+--------------+
|+-+---+-+---+-+|+-+---+-+---+-+|
||1|1 1|1|1 1|1|||0|1 1|0|1 1|3||
|+-+---+-+---+-+|+-+---+-+---+-+|
+--------------+--------------+
```

3.2 Representação de um Caminho

O caminho é representado por uma matriz de mesma forma do puzzle $n \times m$ com os valores em $\{0, 1\}$. O elemento na posição (i, j) da matriz que guarda o valor b representa a aresta $(xy^{-1}(i, j), b)$, onde xy^{-1} é a "função inversa" das funções x e y, ou seja, $xy^{-1}(i, j) = v$ quando $x(v) = i$ e $y(v) = j$. No nosso caso, $xy^{-1}(i, j) := i \cdot m + j$.

Essa representação é muito interessante, pois "a menos de uma função de reorganização" (no nosso caso, a "reorganização vazia") o caminho é uma possível representação direta da solução. Veja a representação da solução do puzzle 1b:

```
    NB. Isso é só para guardar numa variável e mostrar o resultado
    ] b =: 0 1 0 1 0 , 1 1 1 1 1 , 1 1 1 1 1 , 0 1 1 1 0 ,: 0 0 1 0 0
0 1 0 1 0
1 1 1 1 1
1 1 1 1 1
0 1 1 1 0
0 0 1 0 0
    NB. 'X' é preto e '.' é branco
    b { '.X'
.X.X.
XXXXX
XXXXX
.XXX.
..X..
```

3.3 Função Objetivo

A *função objetivo* escolhida transforma o caminho em uma matriz de lista de tamanho dos blocos, de forma que, se essa fosse a solução retornaria uma matriz idêntica a dica. Então, as listas são pareadas e elemento a elemento (possívelmente estendendo a lista menor com 0s) são feitas subtrações seguidas de absoluto (para remover negativos). Somamos todos os números de todas as listas. A função que faz isso é `flist`.

Um exemplo comparando as segundas linhas dos puzzles 1b e 1c:

```
    5 ; 1 1          NB. em representação de blocos
+-+---+
|5|1 1|
+-+---+
    5 0 ,:!.0 1 1    NB. estendendo a menor com 0s
5 0
1 1
    | 5 0 - 1 1      NB. subtraindo, seguido de absoluto
4 1
    +/ 4 1           NB. somando (falta as outras linhas e colunas)
5
```

Com essa escolha de função, procuramos minimizar a sua imagem. Uma propriedade interessante dela é que apenas uma solução vai ter o valor de 0, permitindo parar mais cedo.

3.4 Representação do Feromônio

A lista de feromônios é representada por uma matriz $2 \times n \times m$, no qual o elemento (b, i, j) armazena a quantidade de feromônio da aresta $(xy^{-1}(i, j), b)$, usando o xy^{-1} descrito anteriormente.

Colocar o b na primeira dimensão é vantajoso pela semântica da linguagem. Essa representação permite pegar uma matriz que representa um caminho c e "colar em cima"

sua matriz com elementos complementares ($x \longmapsto 1 - x$), montando as respostas da função $in(c, e)$ (uma "máscara") necessárias para atualizar o feromônio. Também facilita na geração de caminhos, explicada mais a frente.

3.5 Construção de Caminhos

Não foi escolhida nenhuma heurística para a construção dos caminhos, o principal motivo é simplificar a implementação dado que estou fazendo o trabalho sozinho e não ter pensado em uma heurística fácil de implementar.

Dessa forma, não precisamos dos parâmetros α e β, e a escolha do caminho, pelo método da roleta, é influenciada apenas pelo feromônio. Isso, junto a escolha do grafo, faz com que a escolha da cor de um pixel seja completamente independente da escolha da cor de qualquer outro pixel. O esperado é que seja necessário mais iterações para que se ache um caminho bom.

3.6 Condição de Parada

A condição de parada é a combinação de um contador de iterações com uma verificação da imagem da *função objetivo*, se algum deles for 0 podemos parar.

`J` possui uma função de "exponenciação de funções", que aplica a função a quantidade de vezes que o "expoente" indica, de forma que para um n inteiro:

$$f^n(x) := \begin{cases} f^{n+1}(f^{-1}(x)) & , \ n < 0 \\ x & , \ n = 0 \\ f^{n-1}(f(x)) & , \ n > 0 \end{cases}$$

Não precisa se preocupar com o inverso da função. O interessante é que está definida para $n = \infty$, aplica-se f até o resultado não se alterar, efetivamente calculando um *ponto fixo* da função. Na implementação, o loop principal retorna a mesma entrada caso uma das condições indique para parar, permitindo usar essa ideia de *ponto fixo*.

O número máximo de iterações poderia ser n (o expoente), no lugar de ∞, mas deixá-lo como um parâmetro permite uma `main` mais simples.

4 Resultados

Na seção 8, tem um link com instruções de instalação. A versão usada nesse trabalho foi a `J903`, mas não deve ter grandes problemas se a versão for diferente.

Vamos seguir assumindo que já estamos em um sessão (`jconsole`, `jqt` e `jhs` são exemplos), e guardamos o caminho até o `main.ijs` na variável `path`. Então, chamamos o verbo `load` para carregar as funções que estão no arquivo `main.ijs`. Por exemplo:

```
    NB. ']' é só para mostrar o valor da variável
    ] path =: '~/path/to/main.ijs'
~/path/to/main.ijs
    load path
    NB. Note que: load '~/path/to/main.ijs'
    NB. também funciona
```

4.1 Como rodar

A função `main` recebe uma lista encaixotada de *parâmetros*, vamos guardar em `p`, e uma lista encaixotada de *variáveis de loop*, vamos guardar em `v`.

Os *parâmetros* são, em ordem: *taxa de evaporação* ρ, *função objetivo* f e *número de formigas por iteração* $ants$. O exemplo assume que $\rho = 0.4$, f é está otimizando para o puzzle 1a e $ants = 100$.

```
    face
0 0 0 1 0
0 1 0 0 1
0 0 0 0 1
0 1 0 0 1
0 0 0 1 0
    n2b face    NB. calcula as dicas a partir de uma solução
+-+---+-+---+-+
|1|1 1|1|1 1|1|
+-+---+-+---+-+
|0|1 1|0|1 1|3|
+-+---+-+---+-+
    ] p =: (<0.4)`((n2b face)&f)`(<100)
+---+---------------------------+---+
|0.4|+-+-----------------------+|100|
|   ||&|+---------------------+-+|   |
|   || ||+-+-----------------+|f|||   |
|   || |||0|+-+---+-+---+-+|| |||   |
|   || ||| ||1|1 1|1|1 1|1||| |||   |
|   || ||| |+-+---+-+---+-+|| |||   |
|   || ||| ||0|1 1|0|1 1|3||| |||   |
|   || ||| |+-+---+-+---+-+|| |||   |
|   || ||+-+-----------------+| |||   |
|   || |+---------------------+-+||   |
|   |+-+-----------------------+|   |
+---+---------------------------+---+
    NB. Essa caixa do meio é a representação da função objetivo
```

As *váriavei de loop* são, em ordem: *contador de iterações* n que conta quantas iterações ainda pode seguir, *lista de feromônio* τ, *imagem do caminho* dummy e *caminho* dummy, um caminho todo em branco para servir como um *"placeholder"*. Elas podem ser criadas com uma função auxiliar `init`. O exemplo assume que $n = 200$ e que a τ começa com 1 em cada aresta.

```
    NB. não precisa ser a solução,
    NB. basta uma matriz com o tamanho igual
    ] fer =: 1 newfer face
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1

1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
   ] v =: 200 (p init) fer
+---+---------+--+---------+
|200|1 1 1 1 1|14|0 0 0 0 0|
```

```
|   |1 1 1 1 1|  |0 0 0 0 0|
|   |1 1 1 1 1|  |0 0 0 0 0|
|   |1 1 1 1 1|  |0 0 0 0 0|
|   |1 1 1 1 1|  |0 0 0 0 0|
|   |         |  |         |
|   |1 1 1 1 1|  |         |
|   |1 1 1 1 1|  |         |
|   |1 1 1 1 1|  |         |
|   |1 1 1 1 1|  |         |
|   |1 1 1 1 1|  |         |
+---+---------+--+---------+
```

O algoritmo propriamente dito, está implementado em `loop`, mas a `main` coleta mais informações e é mais fácil de usar.

A função `main` pode ser chamada com os argumentos `p` pela esquerda e `v` pela direita. Antes de chamá-la, pode-se alterar a semente inicial se achar interessante. `main` retorna uma lista de caixas, a primeira caixa contém a semente inicial com o estado do gerador de números aleatórios antes de rodar o algoritmo, mas esse estado é gigantesco, então não vai ser mostrado no exemplo; a segunda o tempo de execução; a terceira é uma variável interna que indica se deve parar, já que ele retornou algo, deve ser sempre 0; a quarta são as *variáveis de loop*: n, τ, $f(c^*)$, c^*.

```
    NB. Escolhendo uma semente (opcional)
    wseed 16807

    'rng time stop ret' =: p main v
    time ; stop ; <ret
```

```
+-------+-+-------------------------------------------------------------+
|2.03215|0|+-+-----------------------------------------+-+---------+|
|       | ||0| 1.32328 0.929616 0.620616   1.4589 0.859697|4|1 1 0 1 0||
|       | || |0.784592  1.43173  1.20543  1.51852  1.61536| |1 1 1 1 1||
|       | || | 1.36438  1.78557 0.427282  1.19983  1.08449| |1 1 0 0 0||
|       | || |0.632997 0.864541 0.512813 0.708474  1.00561| |0 0 0 1 0||
|       | || |0.453377 0.409504  1.34556 0.847346  1.18167| |0 1 1 1 0||
|       | || |                                           | |         ||
|       | || |0.676724  1.07038  1.37938 0.541097   1.1403| |         ||
|       | || | 1.21541 0.568269 0.794566 0.481477 0.384644| |         ||
|       | || |0.635621 0.214428  1.57272 0.800168 0.915508| |         ||
|       | || |   1.367  1.13546  1.48719  1.29153 0.994391| |         ||
|       | || | 1.54662   1.5905 0.654442  1.15265 0.818328| |         ||
|       | |+-+-----------------------------------------+-+---------+|
+-------+-+-------------------------------------------------------------+
```

```
    NB. Se não quiser guardar a saída em variáveis pode fazer:
    NB. }. p main v
    NB. para jogar fora o 'rng', ou
    NB. 1 3 { p main v
    NB. para escolher quais resultados quer ver
    NB. (nesse caso 'time' e 'ret')
```

4.2 Método

Para cada puzzle e configuração de parâmetros escolhidos, vai setar uma seed, nesse caso 16807, (isso é para poder reproduzir com mais facilidade) e vão ser executados 5 vezes, assim a tabela irá mostrar os tempos e imagens de cada execução. Para cada tabela, vão aparecer

3 imagens: a solução, a diferença e um melhor caminho encontrado em todas as execuções. A diferença tem duas cores extras: vermelho e verde. Vermelho aponta um pixel colorido erroniamente, enquanto verde aponta um pixel que não foi pintado.

O puzzle escolhido foi 1a. Rodar várias vezes, salvar as imagens e montar a tabela são tarefas repetitivas e consomem um tempo considerável. Não tive tempo sobrando para automatizá-las, idealmente, teria tabelas de todos os puzzles.

Os 3 puzzles da figura 2, forams gerados aleatóriamente para cada um dos seguintes tamanhos: 10×10 (2a), 25×25 (2b) e 50×50 (2b). A ideia seria ter uma noção de o quão bem ele se comporta com puzzles maiores.

(a) Puzzle 10×10 (b) Puzzle 25×25 (c) Puzzle 50×50

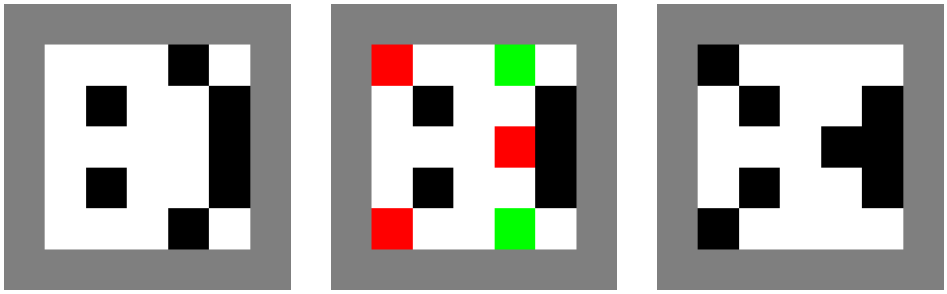
Figura 2: Puzzles maiores

As imagens estão na pasta `images/face/params/`, onde `params` são os parâmetros escolhidos para a execução. Tive problemas em tirar as fotos das respostas, então não tenho certeza se todas estão certas.

4.3 Tabelas

		Evaporação					
Iterações		$\rho = 0.05$		$\rho = 0.25$		$\rho = 0.45$	
$ants = 2500$		**Tempo**	**Imagem**	**Tempo**	**Imagem**	**Tempo**	**Imagem**
100	1	24.17 s	4	24.01 s	4	24.18	4
	2	24.13 s	6	24.08 s	6	24.27	4
	3	24.02 s	6	23.90 s	6	24.14	6
	4	24.07 s	4	23.98 s	6	24.18	6
	5	24.16 s	4	23.90 s	4	24.16	6
200	1	47.71 s	4	47.63 s	4	47.94	4
	2	47.69 s	4	47.84 s	4	47.89	4
	3	47.80 s	4	47.53 s	4	46.51	6
	4	47.85 s	4	47.90 s	4	47.79	4
	5	47.87 s	4	47.99 s	6	47.82	6
400	1	94.91 s	4	95.11 s	4	95.80	4
	2	94.79 s	6	95.23 s	6	95.71	6
	3	94.83 s	4	95.30 s	4	95.81	4
	4	95.07 s	6	95.20 s	4	96.00	4
	5	95.21 s	4	95.69 s	6	95.82	6

Tabela 1: Execuções do Puzzle 1a



(a) Solução (b) Diferença (c) Melhor caminho

Figura 3: Melhor resultado

5 Possíveis Melhorias de Performance

As sugestões de melhorias são para melhorar a qualidade do algoritmo, não necessariamente o tempo de execução.

5.1 Representação do Grafo

Atualmente a distribuição de feromônio no Grafo não guarda de nenhuma forma "da onde" a formiga veio. Talvez uma representação que "se lembre melhor" ajude.

5.2 Função Objetivo

5.2.1 Quadrado em vez de Absoluto

A *função objetivo* atual usa absoluto, talvez elevar ao quadrado dê mais valor a candidatos mais próximos a resposta.

5.2.2 Função mais inteligente

A *função objetivo* atual converte o caminho em uma dica, uma representação que perde informação. Talvez haja uma forma melhor de medir proximidade da solução.

5.3 Geração de Caminhos

Não é usada nenhuma heurística para auxiliar a geração de caminhos. Um simples filtro de caminhos que quebram a dica deve ajudar bastante.

6 Possíveis Melhorias de Funcionalidade

6.1 Permitir para diferentes linhas e colunas

O código só funciona para puzzles quadrados, não deve ser muito difícil ajeitá-lo para aceitar puzzles com dimensões retangulares.

6.2 Generalizar para mais cores

Semelhante ao anterior, não deve ser muito difícil.

6.3 Permitir solução parcial

Isso é algo bem legal, pois permitiria resolver um puzzle parcialmente resolvido (talvez depois de uma série de iterações lógicas). Provavelmente daria um pouco mais de trabalho.

7 Análise final

Achei que os caminhos ficaram bem próximos, mas aumentando o número de iterações não ajudava em chegar mais próximo da resposta. Deu a impressão de que ficou preso em um mínimo local próximo à solução.

Achei que J cobriu bem as necessidades desse trabalho, mas como é o meu primeiro projeto maior com essa linguagem, acabei demorando mais do que o esperado.

Tive problemas em tirar as fotos das respostas, então não tenho certeza se todas estão certas.

Eu acho que seria legal transformar esse trabalho em um paper ou algo assim, mas acho que não tenho muita disponibilidade de tempo para isso. Então deveria ser algo pequeno.

8 Bibliografia

Sobre *Trabalho*:

- Repositório:
  https://github.com/Kiyoshi364/picross-ant-colony-j

Sobre J:

- Página inicial:
  https://wiki.jsoftware.com/wiki/Main_Page
- Instruções de instalação:
  https://wiki.jsoftware.com/wiki/System/Installation
- Dicionário de primitivas:
  https://wiki.jsoftware.com/wiki/NuVoc
- J-playground (uma sessão online):
  https://jsoftware.github.io/j-playground/bin/html/index.html

Sobre *Picross*:

- History of Picross (Nonograms):
  https://youtu.be/qJCPxyi5x5g
- Solving Colored Nonograms:
  https://run.unl.pt/bitstream/10362/2388/1/Mingote_2009.pdf
- NP Completeness of Nonograms:
  https://kelbybsandvick.github.io/pdf/NPProblemPaper.pdf
- Complexity and solvability of Nonogram puzzles:
  https://fse.studenttheses.ub.rug.nl/15287/1/Master_Educatie_2017_RAOosterman.pdf
- kamilkhanlab/nonogram-ilp (Implementação de solver em GAMS):
  https://github.com/kamilkhanlab/nonogram-ilp
- Nonogram Wikipédia:
  https://en.wikipedia.org/wiki/Nonogram