Uma meta-heurística eficiente para o problema de inundação em grafos

André Renato Villela da Silva
Universidade Federal Fluminense
Instituto de Computação
Departamento de Computação
R. Recife s/n, Jardim Bela Vista - Rio das Ostras/RJ CEP 28895-532
avillela@ic.uff.br

RESUMO

Este trabalho aborda o Problema de Inundação em Grafos, cujo objetivo é tornar monocromático um grafo colorido em vértices com o menor número possível de inundações. Por inundação, entende-se a mudança da cor de um vértice pivô para uma cor c, o que faz com que esse vértice seja agregado a todos os vértices vizinhos que tenham a mesma cor c. A literatura do problema apresenta quatro algoritmos heurísticos e uma formulação matemática. Um algoritmo evolutivo é proposto com o objetivo de produzir soluções de melhor qualidade em tempo computacional pequeno. Os resultados mostram que o algoritmo evolutivo é realmente eficiente mesmo consumindo apenas poucos segundos.

PALAVRAS CHAVE. Problemas de inundação em grafos, meta-heurísticas, algoritmos evolutivos

Área Principal: Meta-heurísticas

ABSTRACT

This work deals with the Flooding Problem in Graphs, whose goal is to make monochromatic a vertex-colored graph with the smallest possible number of flooding operations. Flooding means changing the color of a pivot vertex to a color c , which makes this vertex united to all neighboring vertices that have the same color c . The literature of the problem presents four heuristic algorithms and a mathematical formulation. An evolutionary algorithm is proposed in order to produce better quality solutions in small computational time. The results show that the evolutionary algorithm is really efficient even consuming only a few seconds.

KEYWORDS. Graph flooding problems, metaheuristics, evolutionary algorithms

Main Area: Metaheuristics

1. Introdução

O Problema de Inundação em Grafos é derivado de um popular videojogo chamado Flood-It [LabPixies, 2015]. O jogo é composto por uma matriz de tamanho $N \times M$, na qual cada elemento é representado por uma cor. As chamadas regiões monocromáticas são formadas por elementos adjacentes (apenas na vertical e/ou horizontal) que sejam da mesma cor. Essas regiões se comportam como se fossem um único elemento. O objetivo do jogo é, com o menor número possível de etapas, tornar toda a matriz monocromática através da mudança de cor de uma região inteira de cada vez. Sempre que uma região mudar para uma cor c, ela se une a todas as eventuais regiões adjacentes de mesma cor c. Esse processo chama-se inundação. A Figura 1 mostra a sequência de inundações que tornam monocromática a matriz em questão.

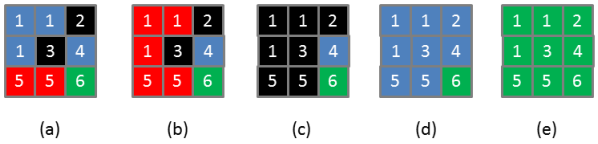


Figura 1: Sequência de inundações que tornam a matriz monocromática: (a) inicialmente a matriz apresenta 6 regiões monocromáticas, sendo a região 1 fixa (pivô); (b) escolhida a cor vermelha, o que inunda a região 5; (c) escolhida a cor preta, inundando também as regiões 2 e 3; (d) escolhida a cor azul; (e) escolhida a cor verde, tornando toda a matriz monocromática em 4 etapas.

O jogo Flood-It possui uma versão fixa, onde apenas uma das regiões monocromáticas pode mudar de cor inicialmente. As demais só poderão mudar, juntamente com a primeira, quando forem inundadas. Na outra versão denominada livre, qualquer uma das regiões pode ser recolorida a qualquer momento. Contudo o objetivo de ambas é o mesmo: tornar a matriz monocromática com o menor número possível de etapas (inundações). Também é possível estabelecer uma versão de decisão para o problema, definindo um número máximo Q de etapas para se cumprir o objetivo. O problema pode ser formulado da seguinte maneira: "é possível tornar a matriz monocromática em no máximo Q inundações?". O problema em sua forma de minimização é NP-difícil. Portanto, não são conhecidos algoritmos eficientes para encontrar a solução ótima em tempo aceitável.

Embora as apresentações mais comuns do problema sejam na forma matricial, computacionalmente é mais eficiente trabalhar com estruturas de grafos. Nesse sentido, cada vértice do grafo corresponde a uma região monocromática da matriz. As arestas representam a relação de adjacência entres essas regiões. Pode-se, portanto, definir o problema utilizando esta interpretação por meio de grafos. Seja $G = (V, A)$, um grafo não-direcionado contendo um conjunto V de vértices coloridos e um conjunto A de arestas entre estes vértices, deseja-se tornar o grafo inteiramente monocromático com o menor número possível de inundações. Uma inundação pode ser descrita como o procedimento de mudança de cor de um vértice v para uma cor arbitrária c. Se existirem vértices adjacentes a v com a mesma cor c, estes vértices podem ser considerados unidos entre si (inundados) e passam a mudar de cor conjuntamente. Este processo cria uma espécie de "efeito-dominó" na coloração dos vértices do grafo. Desta forma, o objetivo do problema é fazer com que todos os vértices do grafo tenham a mesma cor final (qualquer que seja) com o menor número possível de inundações. Na versão fixa, existe apenas um vértice pivô considerado inundado desde o início do problema. Assim, todos os vértices precisam ser inundados a partir deste vértice pivô. Na versão livre, não existe este vértice pivô, ou seja, qualquer vértice pode mudar de cor a cada etapa.

O restante deste trabalho se divide da seguinte forma: na Seção 2 são apresentados alguns resultados de trabalhos anteriores da literatura. Na Seção 3, uma versão de um Algoritmo Evolutivo é proposto para o problema, na versão fixa. Os resultados computacionais dos experimentos realizados são mostrados na Seção 4. A Seção 5 apresenta as conclusões deste trabalho e sugestões de trabalhos futuros.

2. Revisão da literatura

O jogo em questão pode ser utilizado como modelo para algumas aplicações reais, como propagação de doenças descritas em [Adriaen et al., 2004], que podem ocorrer de forma similar ao jogo. Em [Souza et al., 2013], é apresentada uma analogia do problema de inundação em árvores com subcaso do problema da supersequência comum mais curta.

Em [Arthur et al., 2010], os autores mostraram que as duas versões (fixa e livre) para matrizes $N \times N$ coloridas com mais de três cores são NP-difíceis. Em [Meeks e Scott, 2013], foi mostrado que a versão livre pode ser resolvida em tempo polinomial para matrizes $1 \times N$. As duas versões permanecem NP-difíceis para matrizes $3 \times N$ coloridas com pelo menos quatro cores. Entretanto, nestas mesmas matrizes o problema permanece em aberto se forem utilizadas apenas três cores.

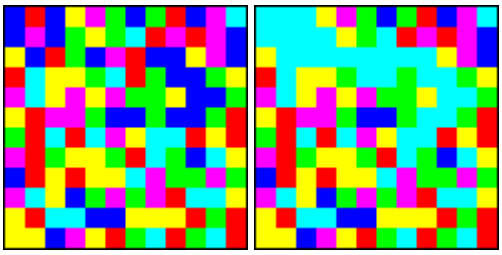
Para matrizes $2 \times N$, na versão fixa do problema, foi apresentado um algoritmo de tempo polinomial em [Clifford et al., 2012]. Para estas matrizes, a versão livre permanece NP-difícil, podendo ser resolvida em tempo polinomial em grafos 2-coloridos, de acordo com [Meeks e Scott, 2011]. Em [Lagoutte, 2010], mostrou-se que a versão livre é polinomialmente solucionável em ciclos e que as duas versões são NP-difíceis em árvores.

Também existem algoritmos de tempo polinomial para os seguintes casos, como mostrado em [Souza et al., 2014]: grades circulares $2 \times N$, a segunda potência de um ciclo com n vértices, d-tabuleiros $2 \times N$ (grades $2 \times N$ onde a d-ésima coluna é monocromática). Nestes dois últimos casos, a versão livre é NP-difícil.

Em [Barros et al., 2015] são apresentados as primeiras heurísticas construtivas gulosas para o problema e a primeira formulação matemática por meio de um programa inteiro misto. A heurística gulosa básica consiste em escolher, a cada etapa, a cor capaz de inundar o maior número de vértices naquele momento. Outros três algoritmos são baseados no conceito de distância entre um vértice não inundado v e outro vértice já inundado i. A distância entre os dois é equivalente ao tamanho do caminho mais curto P entre esses vértices. A heurística Flooding-I calcula, a cada etapa, o vértice não inundado v com maior distância e realiza as inundações seguindo a ordem de cores do caminho P respectivo. O objetivo é tentar alcançar o mais cedo possível vértices que estejam distantes do vértice pivô.

A heurística Flooding-II é muito semelhante à anterior, porém a utilização das distâncias e dos respectivos caminho se encerra ao inundar $3/4$ dos vértices do grafo. A partir desse ponto, a heurística gulosa é empregada até o fim do algoritmo. Por fim, a heurística Flooding-III é uma adaptação da anterior. As etapas de inundação que decorrem das cores de P são realizadas com exceção dos 4 últimos vértices. Nesse aspecto há uma falha na descrição do algoritmo, pois o mesmo pode simplesmente não terminar. Essa situação ocorre antes da chamada da heurística gulosa, ou seja, quando os $3/4$ dos vértices de grafo ainda não foram inundados. Se todos os caminhos P existentes possuirem tamanho menor ou igual a 4, não fica clara qual deve ser a decisão do algoritmo, já que, em princípio, as últimas 4 etapas de inundação não deveriam ocorrer conforme o caminho P.

A Figura 2 mostra um exemplo em forma matricial onde não fica clara a convergência pretendida. Em Fig. 2(a) temos a matriz original contendo 99 regiões monocromáticas (vértices). Após 8 etapas de inundação conforme as cores dos caminhos P computados, apenas 15 regiões já foram inundadas a partir do pivô (região do canto superior esquerdo). Todas as outras regiões encontram-se a 4 ou menos unidades de distância, porém a heurística gulosa ainda não é utilizada já que não se atingiu o valor de $3/4$ das regiões (vértices). Pela descrição do algoritmo não fica claro se a heurística gulosa deve ser invocada a partir desse momento ou se as cores do caminho P devem ser seguidas até o valor de $3/4$ dos vértices, mesmo que os caminhos tenham menos de 4 unidades de distância. Os resultados apresentados mostraram a heurística Flooding-III um pouco melhor do que a Flooding-II, que por sua vez foi melhor que as outras heurísticas de [Barros et al., 2015].



(a) (b)

Figura 2: (a)Matriz original com 99 regiões monocromáticas (vértices). (b)Matriz obtida após 8 etapas de inundação (rosa, azul escuro, verde, azul escuro, rosa, vermelho, azul escuro, azul claro) conforme a primeira parte do algoritmo Flooding-III.

3. Métodos e Técnicas

Este trabalho apresenta um Algoritmo Evolutivo, denominado EA, para solucionar o problema de Inundação em Grafos. Essa meta-heurística foi escolhida por apresentar excelentes resultados computacionais em diversas áreas da computação como é possível ver em [Silva, 2013, Gonçalves et al., 2016, Yevseyeva et al., 2013].

A representação de um indivíduo consiste em um vetor de $n-1$ números reais. Cada posição está associada a um vértice do grafo, com exceção do vértice pivô, inicialmente já inundado. O valor em cada posição corresponde à prioridade daquele vértice em ter sua cor escolhida quando o vértice puder ser inundado.

Portanto, o algoritmo de criação de um indivíduo recebe uma lista contendo a prioridade de cada vértice. Inicialmente, apenas os vértices adjacentes ao pivô podem ser inundados. O vértice que tiver a maior prioridade é selecionado e sua cor é escolhida para ser utilizada nessa etapa de inundação. A lista de vértices que podem sofrer inundação é atualizada, um novo vértice é selecionado e o processo se repete até que todo o grafo se torne monocromático. A aptidão deste indivíduo é a quantidade de etapas necessárias para inundar completamente o grafo. As cores escolhidas em cada etapa são armazenadas em uma lista, que representa uma solução aproximativa para a instância em questão.

As prioridades dos indivíduos iniciais são geradas de acordo com a seguinte fórmula: $p_i = (cont_i * k + u)/d$, onde p_i é o valor da prioridade do vértice i, $cont_i$ é a quantidade total de vértices com a mesma cor de i, k é um parâmetro de ponderação, u é uma variável aleatória uniforme discreta e d é um parâmetro de escala. Os valores desses parâmetros foram definidos em testes preliminares ($k = 10$, $u = [1, 100]$, $d = 1000$) e visam dar uma prioridade maior aos vértices com as cores mais frequentes, pois inundando-os é possível alcançar uma quantidade maior de vértices posteriormente. A variável u visa criar uma distinção entre dois indivíduos à medida em que, a cada vértice de cada novo indivíduo, ela pode assumir valor diferente. O parâmetro d é o menos necessário e foi utilizado apenas para que as prioridades ficassem no intervalo real $(0, 1)$. Os testes preliminares também adotaram a criação de prioridades totalmente aleatórias, mas as soluções obtidas foram um pouco piores. Um pseudo-código de EA pode ser visto no Algoritmo 1.

A população inicial (linha 1) cria um total de $totalIndiv$, mas somente os $maxIndiv$ melhores permanecem na população. O critério de parada é determinado na linha 3. A população de indivíduos filhos (linha 4) é gerada por um operador de recombinação que tem um valor de λ como parâmetro. Quanto maior o valor do parâmetro, mais diferentes podem ser os filhos em relação a seus pais. Na linha 5, as populações de indivíduos pais e filhos são misturadas e os $maxIndiv$ melhores indivíduos permanecem na população seguinte. A cada F gerações (linhas

Algoritmo 1 Pseudo-código do Algoritmo EA

```
1:  pop ← PopulacaoInicial(totalIndiv,maxIndiv)
2:  geracao ← 1
3:  melhorSol ← MelhorIndividuo(pop)
4:  while tempo total não excedido do
5:      filhos ← Recombinacao(pop,λ)
6:      pop ← SelecaoNatural(pop+filhos,maxIndiv)
7:      melhorSol ← MelhorIndividuo(pop)
8:      geracao ← geracao + 1
9:      if geracao (mod F) = 0 then
10:         incrementa(λ,incLambda,maxLambda)
11:     end if
12: end while
13: return melhorSol
```

9 e 10), o valor de λ é incrementado por λ_{inc} (até o limite de λ_{max}) para permitir que indivíduos filhos cada vez mais diferentes possam surgir. Os valores definidos em testes preliminares são: $totalIndiv = 1000$, $maxIndiv = 50$,$F = 100$, $\lambda_{inc} = 0,2$ e $\lambda_{max} = 1,0$. O valor inicial de λ foi definido em 0,1.

O parâmetro λ é utilizado no operador de recombinação. Este operador é responsável por gerar uma população de indivíduos filhos recombinados a partir da população atual de pais. A população é dividida em três classes: a classe A é composta pelos 10% melhores indivíduos; a classe C é composta pelos 60% piores indivíduos e a classe B pelos 30% restantes. O Algoritmo 2 mostra um pseudo-código da recombinação utilizada. São escolhidos aleatoriamente um pai proveniente da classe A e outro da classe B (linhas 1 e 2). Dois indivíduos filhos são gerados, tendo como valor base das prioridades, para cada vértice, a média das prioridade dos pais (linha 4). Para que os dois filhos não sejam absolutamente iguais, o valor de λ é multiplicado por uma variável contínua y do intervalo $[-1, +1]$, uniformemente distribuída (linhas 5 e 6). Desta forma, as prioridades finais dos filhos podem ficar um pouco acima ou abaixo da média já calculada. Quanto maior o valor de λ mais distintos os filhos podem ser entre si. Com o passar das gerações, a tendência é que os indivíduos de uma mesma população sejam cada vez mais semelhantes. Um valor maior de λ pode permitir que um filho gerado pela recombinação se diferencie bastante dos seus pais.

Algoritmo 2 Pseudo-código do operador de recombinação

```
1: pai1 ← EscolhePai(pop,ClasseA)
2: pai2 ← EscolhePai(pop,ClasseB)
3: for i=2..n do
4:     media = (pai1[i]+pai2[i])/2
5:     filho1[i] = media*(1+y*λ)
6:     filho2[i] = media*(1+y*λ)
7: end for
8: return filho1,filho2
```

4. Resultados Computacionais

Os testes computacionais utilizaram o seguinte equipamento: laptop com processador Intel I7-3630QM, 8GB de memória RAM, sistema operacional Windows 8.0. O código foi escrito em linguagem C++ e compilado com o GNU GCC 4.7.1. Em [Barros et al., 2015], foram utilizadas instâncias compostas por matrizes quadradas $N \times N$, sendo N=14 e 6 cores disponíveis. Infelizmente, essas instâncias não foram disponibilizadas. Para os experimentos deste trabalho, foram geradas 5 instâncias com $N = 12, 15, 20$ e 6 cores. As cores de cada posição das matrizes foram geradas com probabilidade uniforme. Também foram geradas mais 3 instâncias adicionais com $N = 20$ e 7 ou 8 cores, respectivamente. Elas podem ser visualizadas na Figura 3.

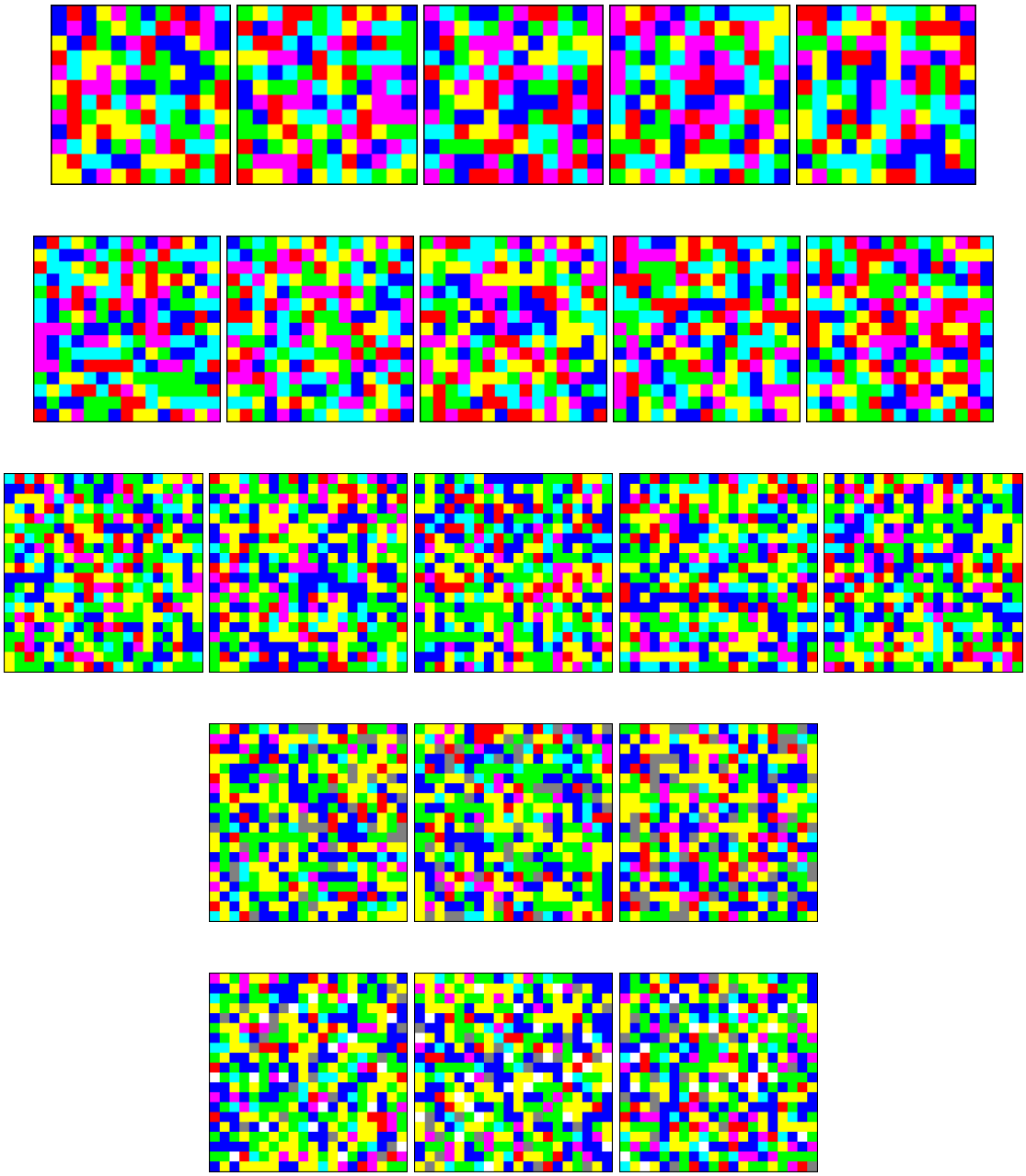


Figura 3: Instâncias utilizadas nos testes computacionais. Na primeira linha, estão as 5 instâncias com $N = 12$ e $C = 6$. Na segunda linha, as instâncias com $N = 15$ e $C = 6$. Na terceira linha, as instâncias com com $N = 20$ e $C = 6$. Por fim, nas últimas linhas, as 3 instâncias com $N = 20$ e $C = 7$ ou $C = 8$, respectivamente.

O primeiro teste considera principalmente o valor da solução obtida por cada algoritmo. De [Barros et al., 2015], foram considerados os algoritmos Flooding-I e Flooding-II por apresentarem convergência e por terem obtidos os melhores resultados. O algoritmo Flooding-III não foi considerado, pois, como já explicado anteriormente, há casos onde a descrição do algoritmo não permite estabelecer como efetivamente é sua convergência. O otimizador CPLEX 12.6.1 foi utilizado juntamente com a formulação matemática também proposta em [Barros et al., 2015]. O algoritmo EA proposto neste artigo foi executado 30 vezes, enquanto os demais foram executados uma única vez por serem configurados de forma determinística.

A Tabela 4 apresenta os principais resultados deste experimento. As quatro primeiras colunas indicam as características das instâncias, onde N representa a dimensão da matriz quadrada $N \times N$, C indica o número de cores distintas existentes na instância, $\#$ é apenas um identificador da instância em questão e $|V|$ indica o número de vértices da instância, após a conversão da matriz para grafo. Assim, a primeira instância da Tabela, por exemplo, é derivada de uma matriz quadrada 12×12, com 6 cores e possui 99 vértices na forma de grafo. Essa instância é a representada também na Fig. 2(a).

Instância				[Barros et al., 2015]		EA			CPLEX		
N	C	#	|V|	Flood.I	Flood.II	Mín.	Freq.	Média	Sol.	Tempo(s)	Lim. Inf
12	6	1	99	26	26	18	18	18,4	18	324,17	
12	6	2	105	24	24	20	12	20,6	20	64,80	
12	6	3	97	20	20	18	20	18,4	18	18,77	
12	6	4	101	21	21	18	27	18,1	18	36,03	
12	6	5	92	18	19	16	27	16,1	16	52,19	
15	6	1	146	25	24	19	13	19,6	19	882,45	
15	6	2	167	30	30	24	2	25,5	24	14400,00	22
15	6	3	142	27	27	20	3	21,1	20	1135,08	
15	6	4	146	23	23	19	10	19,8	19	162,03	
15	6	5	164	28	28	23	1	24,5	23	3187,61	
20	6	1	266	29	28	25	1	27,1	25	14400,00	24
20	6	2	251	30	29	22	15	22,8	22	5151,72	
20	6	3	266	35	33	28	4	29,5	29	14400,00	26
20	6	4	269	27	27	25	2	29,6	25	14400,00	24
20	6	5	253	31	31	28	13	28,6	27	14400,00	26
20	7	1	251	36	37	25	2	26,0	26	14400,00	24
20	7	2	243	31	31	25	23	25,3	25	14400,00	24
20	7	3	265	32	34	26	3	27,4	25	14400,00	23
20	8	1	267	36	36	28	20	28,3	27	14400,00	24
20	8	2	258	29	30	24	14	24,6	27	14400,00	20
20	8	3	298	33	34	31	7	31,9	30	14400,00	25

Tabela 1: Principais resultados comparativos entre os algoritmos testados.

Os tempos de processamento dos algoritmos Flooding-I e II variaram entre 0,001 e 0,016 segundos. O limite de tempo do CPLEX foi estabelecido em 4 horas (14400 segundos). Nas instâncias nas quais o CPLEX não terminou de executar nesse prazo, ficaram registrados o valor da solução incumbente e do limite inferior (dual). O tempo máximo para o EA foi equivalente a $|V|/10$ segundos em cada execução. Sobre a formulação matemática testada, é importante destacar que o horizonte de planejamento para as etapas de inundação adotado em [Barros et al., 2015] é muito exagerado, sendo equivalente a $|V|$. No pior caso, apenas um vértice do grafo seria inundado em cada etapa, sendo necessárias $|V| - 1$ etapas para resolver o problema. Embora este horizonte

possa parecer razoável, muitas variáveis são consideradas desnecessariamente na modelagem, o que faz com que a execução consuma muito mais tempo e memória. Neste artigo, foi utilizado como horizonte de planejamento o valor da solução obtida pelo método Flooding-II. Isto se justifica porque se é possível encontrar uma solução heurística para o problema com t etapas, a solução ótima deve ter $t* \leq t$ etapas. Isto reduz consideravelmente o número de variáveis e permite uma execução muito melhor da modelagem proposta. Como exemplo, na instância (N=20;C=6;#=4), o horizonte é reduzido de 269 etapas para apenas 27, o que equivale a eliminar aproximadamente 90% das variáveis originalmente presentes na modelagem.

Um segundo ponto a se destacar é que as heurísticas Flooding-I e II obtiveram resultados bem semelhantes, o que confirma em parte os resultados apresentados também em [Barros et al., 2015]. Em relação ao algoritmo EA, pode-se perceber que a média das melhores soluções não ficou muito distante da melhor solução encontrada ao longo das 30 execuções (coluna Mín). Em algumas instâncias, o número de vezes em que essa melhor solução foi obtida (coluna Freq.) é bastante alto. Em outras, porém, essa melhor solução foi obtida poucas vezes. O algoritmo EA pode ser considerado razoavelmente robusto, levando-se em consideração a média das soluções. No tocante à frequência da melhor das soluções, ainda é possível melhorá-lo um pouco, provavelmente com a confecção de buscas locais mais intensivas.

Outro resultado interessante pode ser visto na comparação do EA com o CPLEX. Em todas as 10 instâncias nas quais o CPLEX terminou de executar no prazo de 4 horas, o EA obteve, pelo menos uma vez, a solução ótima. Nas 11 instâncias cujo valor do ótimo ainda está em aberto, o EA obteve resultado pior do que o CPLEX em apenas quatro delas. Por outro lado, o valor do EA foi melhor do que a solução incumbente do CPLEX em 3 instâncias. Esses valores são ótimos para uma meta-heurística relativamente simples como o EA e mostram seu grande potencial em encontrar soluções de boa qualidade. Vale a pena destacar a instância (N=20;C=8;#=2), cujo *gap* entre os limites superior e inferior foi o maior do todos. Nessa instância, o valor do EA foi mais de 10% melhor que o CPLEX. Por outro lado, a formulação apresentada em [Barros et al., 2015] começa a apresentar desempenho muito ruim conforme o número de vértices da instância se aproxima de 250, o que reforça a necessidade de boas meta-heurísticas para tratar do problema. É importante destacar que durante sua execução o CPLEX utilizou os oito núcleos do processador, enquanto os demais algoritmos apenas um núcleo por serem essencialmente algoritmos sequenciais.

Uma comparação rápida entre os algoritmos Flooding-I, II e o EA revela que os primeiros são muito rápidos, mas ficam razoavelmente longe da melhor solução do EA. No problema em questão, não existe a necessidade de uma solução *on-line*, portanto, consumir um pouco mais de tempo para prover uma solução melhor parece mais interessante do ponto de vista prático. Em algumas instâncias, a melhor solução do EA foi mais de 20% melhor do que as dos outros algoritmos.

O segundo experimento computacional visa exatamente analisar o tempo necessário para que o EA apresente uma solução de boa qualidade. Tanto no teste anterior quanto neste, o limite de tempo dado ao EA é de $|V|/10$ segundos. Quatro instâncias foram selecionadas aleatoriamente e, a cada vez que o EA melhora sua solução, é registrado na Figura 4 um novo ponto (X, Y), onde Y representa o valor dessa nova solução e X representa o tempo (em segundos) no qual a solução foi produzida.

É possível perceber que nas duas instâncias menores (Fig.4(a) e (b)), a convergência do EA se deu praticamente em menos de 1,0s. Em outras palavras, utilizar mais de 1,0s para o EA praticamente não melhora em nada a solução final. Nas instâncias maiores (Fig.4(c) e (d)), o mesmo comportamento pode ser visto, porém próximo de 5,0s, o que ainda é um tempo computacional muito pequeno. É importante ressaltar novamente que os resultados existentes na literatura são produzidos por heurísticas gulosas simples, que naturalmente são muito rápidas. No entanto, tais resultados se mostraram muito ruins (Tabela 1) o que justifica o emprego de técnicas mais complexas como as meta-heurísticas propostas. Por analisarem uma grande quantidade soluções diferentes ao longo de sua execução, as meta-heurísticas consomem obviamente um tempo computacional

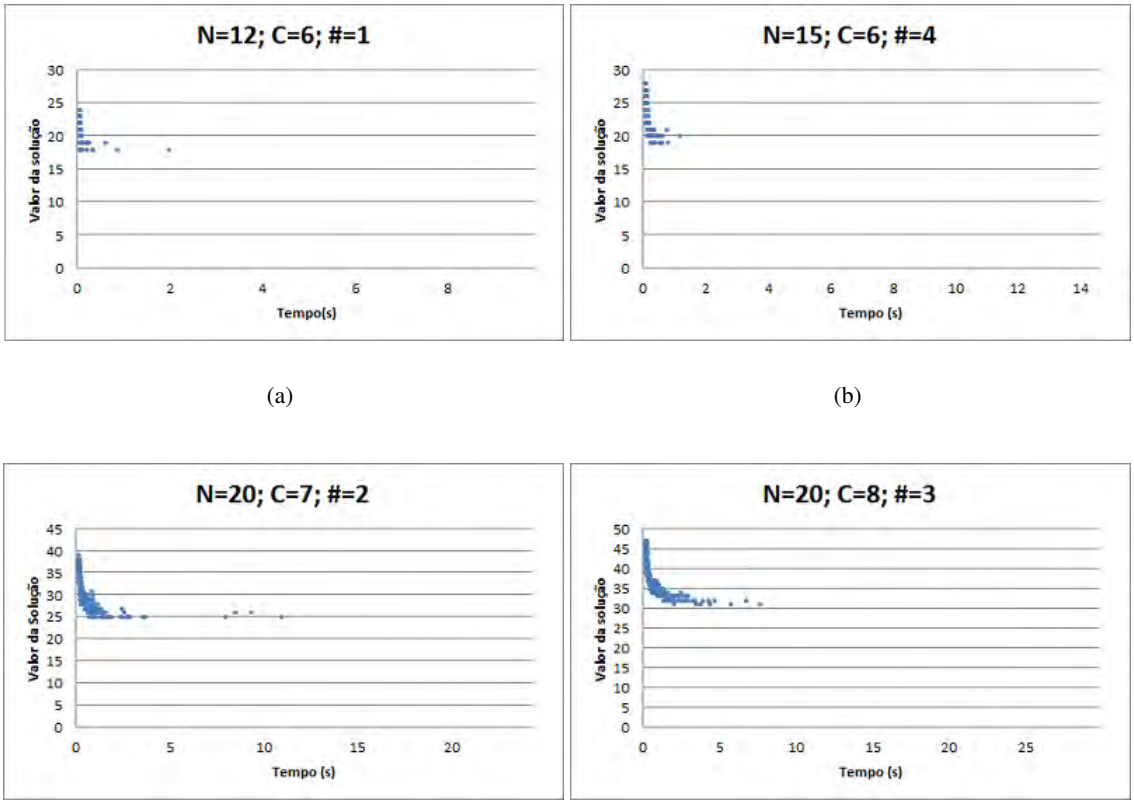


Figura 4: Gráficos de convergência, mostrando cada nova melhor solução obtida pelo EA no total de 30 execuções, para quatro instâncias do problema.

maior, que neste caso, foi bem aproveitado já que suas soluções foram bem melhores. Utilizar o tempo total da heurística gulosa como critério de parada para as meta-heurísticas faz pouco sentido pela diferença existente entre esses paradigmas.

Também foi testada uma maneira diferente para gerar instâncias para o problema. Ao invés do uso de probabilidade uniforme para a escolha individual das cores na confecção da matriz, foi adotada uma estratégia na qual metade das cores disponíveis tiveram probabilidade maior de serem escolhidas. Portanto, as cores disponíveis foram divididas em dois grupos: o grupo prioritário e o grupo regular. A cada nova escolha de cor para gerar a matriz colorida, o grupo prioritário teve 70% de probabilidade de ser escolhido, enquanto o grupo regular teve 30%. Dentro de cada grupo, a cor específica foi escolhida com probabilidade uniforme.

Instância				[Barros et al., 2015]		EA			CPLEX		
N	C	#	|V|	Flood.I	Flood.II	Mín.	Freq.	Média	Sol.	Tempo(s)	Lim. Inf
12	6	1	100	26	26	18	16	18,5	18	186,84	
12	6	2	100	24	24	20	11	20,7	20	106,17	
12	6	3	94	20	20	18	13	18,6	18	33,61	
12	6	4	105	21	21	18	26	18,1	18	99,41	
12	6	5	93	19	18	16	29	16,0	16	55,88	
15	6	1	142	24	25	19	5	20,0	19	625,52	
15	6	2	142	30	30	24	1	25,9	24	14400,00	23
15	6	3	140	27	27	20	1	21,4	20	949,02	
15	6	4	142	23	23	19	14	19,6	19	1561,94	
15	6	5	137	28	28	24	13	24,6	23	639,75	
20	6	1	251	28	29	25	1	27,5	26	14400,00	23
20	6	2	272	29	30	22	7	23,2	22	3817,00	
20	6	3	271	33	35	29	11	29,9	29	14400,00	25
20	6	4	257	27	27	26	1	30,4	25	14400,00	24
20	6	5	258	31	31	28	7	28,8	27	14400,00	26
20	7	1	261	37	36	25	5	26,0	26	14400,00	24
20	7	2	283	31	31	25	12	25,8	29	14400,00	21
20	7	3	264	34	32	26	1	28,2	25	14400,00	24
20	8	1	282	36	36	28	11	28,7	29	14400,00	24
20	8	2	259	30	29	24	3	25,5	27	14400,00	20
20	8	3	266	34	33	31	3	32,5	31	14400,00	25

Tabela 2: Principais resultados comparativos entre os algoritmos testados, para instâncias com prioridade de cores.

A Tabela 4 apresenta o resultados para estas instâncias. Novamente, pode-se perceber que o algoritmo EA teve desempenho muito melhor que as heurísticas gulosas. Comparando as duas formas de gerar as matrizes, não foi possível afirmar que houve uma estratégia na qual as instâncias sejam mais difíceis na prática, pois o desempenho relativo dos métodos entre si foi bastante parecido. Todas as instâncias utilizadas encontram-se em `www.professores.uff.br/asilva/flooding_problems.html`.

5. Conclusão e trabalhos futuros

Este trabalho abordou o Problema de Inundação em Grafos, que consiste em tornar monocromático um grafo colorido em arestas por meio do menor número possível de etapas de inundação. Por inundação, entende-se a mudança da cor de um vértice pivô para uma cor c, o que faz com que esse vértice seja agregado a todos os vértices vizinhos que tenham a mesma cor c. Nas etapas sub-

sequentes, tanto o vértice pivô quanto os seus vértices agregados mudam de cor simultaneamente. A literatura do problema apresenta 4 heurísticas construtivas e uma formulação matemática.

Foi proposto para o problema um algoritmo evolutivo denominado EA. Embora estruturalmente muito simples, o algoritmo EA mostrou-se bem melhor que as heurísticas existentes, encontrando a solução ótima de diversas instâncias em poucos segundos. O algoritmo também mostrou uma rápida convergência e razoável robustez. Esses resultados mostram que o EA é uma técnica heurística muito boa para obtenção rápida de soluções de boa qualidade.

O algoritmo ainda pode ser melhorado para que apresente um pouco mais de robustez, no que se refere a repetir mais vezes o valor da melhor solução que ele pode encontrar, como visto na Tabela 4. Para tanto, acredita-se que a implementação de buscas locais mais intensivas possa ser um bom caminho. Para não afetar tanto o tempo computacional necessário para as buscas, uma possibilidade seria a paralelização do algoritmo, tende um vista que muito sistemas computacionais modernos dispõem de diversos núcleos de processamento que podem operar coordenadamente. A criação de boas buscas locais pode permitir também a elaboração de outras versões meta-heurísticas como GRASP e VND/VNS, por exemplo. Testes com outras distribuições de probabilidade também podem ser interessantes, principalmente em aplicações específicas do problema.

6. Agradecimentos

À FAPERJ, pelo financiamento parcial desta pesquisa.

Referências

Adriaen, M., De Causmaecker, P., Demeester, P., e Vanden Berghe, G. (2004). Spatial simulation model for infectious viral disease with focus on sars and the common flu. In *37th Annual Hawaii International Conference on System Sciences, IEEE Computer Society, ISBN: 0-7695-2056-1*.

Arthur, D., Clifford, R., Jalsenius, M., e ans B. Sach, A. M. (2010). The complexity of flood filling games. In *FUN, volume 6099 of Lecture Notes in Computer Science*, pp. 307–318. Springer, ISBN 978-3-642-13121-9.

Barros, B., Pinheiro, R., e Souza, U. (2015). Métodos heurísticos e exatos para o problema de inundação em grafos. In *Anais do XLVII Simpósio Brasileiro de Pesquisa Operacional (SBPO2015)*.

Clifford, R., Jalsenius, M., Montanaro, A., e Sach, B. (2012). The complexity of flood filling game. *Theory Comput Syst*, 50:72–92.

Gonçalves, J. F., Resende, M. G. C., e Costa, M. D. (2016). A biased random-key genetic algorithm for the minimization of open stacks problem. *International Transactions in Operational Research*, 23:25–46.

LabPixies (2015). Labpixies - the coolest games! `http://www.labpixies.com`. Acessado em 27/12/2015.

Lagoutte, A. (2010). Jeux d'inondation dans les graphes. Technical report, ENS Lyon, HAL: hal-00509488.

Meeks, K. e Scott, A. (2011). The complexity of flood-filling games on graphs. *Discrete Applied Mathematics*, 160:959–969.

Meeks, K. e Scott, A. (2013). The complexity of free-flood-it on 2n boards. *Theoretical Computer Science*, 500:25–43.

Silva, A. (2013). Aplicação de uma meta-heurística não específica em um problema de árvores em grafos. In *Anais do XLV Simpósio Brasileiro de Pesquisa Operacional (SBPO2013)*.

Souza, U., Protti, F., e da Silva, M. D. (2013). Parameterized complexity of flood-filling games on trees. *Lecture Notes in Computer Science*, 7936:531–542.

Souza, U., Protti, F., e da Silva, M. D. (2014). An algorithmic analysis of flood-it and free-flood-it on graph powers. *Discrete Mathematics and Theoretical Computer Science*, 16:279–290.

Yevseyeva, I., Basto-Fernandes, V., Ruano-Ordás, D., e Méendez, J. R. (2013). Optimising anti-spam filters with evolutionary algorithms. *Expert Systems with Applications*, 40(10):4010 – 4021.