Trabalho Prático 1: Cubo Mágico

Computação Natural

Victor Pires Diniz

09 de setembro de 2016

1 Introdução

O Cubo Mágico (Rubik's Cube) é um quebra-cabeça combinatório que se tornou muito famoso na década de 1980. Sua resolução não-intuitiva e o grande número de estados possíveis que o cubo pode assumir o tornaram objeto de diversos estudos matemáticos. Hoje se sabe que um algoritmo ótimo (*God's Algorithm*) para o cubo mágico que resolva qualquer instância com o mínimo número de movimentos possível é capaz de fazê-lo em no máximo 20 movimentos[3].

Enumerar todas as combinações do cubo é inviável - existem 8! arranjos das quinas dos cubos e $\frac{12!}{2}$ arranjos das arestas (peças que contém dois quadrados). Combinando as duas situações e levando em consideração a interação entre as posições das peças de quina e aresta, é possível calcular o número de combinações do cubo mágico:

$$8! * 3^8 * \frac{12!}{2} * 2^{11} \approx 43^{10^{18}}. \tag{1}$$

Evidentemente, força bruta é incapaz de resolver tal problema. Uma forma de tratar a explosão combinatória e enumerar soluções de forma inteligente é utilizar um algoritmo genético: indivíduos, no caso, seriam sequências de movimentos no cubo. Esses indivíduos poderiam ser avaliados por alguma métrica que tente determinar o quão próximo o cubo está da solução após serem aplicados a ele os movimentos.

2 O Problema

Uma configuração do cubo mágico é determinada pela posição das suas peças. Dada uma configuração inicial, é desejado encontrar uma solução que resolva o cubo mágico com o menor número possível de passos. A especificação do trabalho determina que seja utilizado um algoritmo genético para fazê-lo.

3 Notação e noções básicas

O cubo mágico é composto de 54 quadrados (*facelets*), ou, alternativamente, 26 pequenos cubos (*cubelets*), dos quais apenas 45 e 20, respectivamente, podem ser movimentados em relação ao restante do cubo. Esta documentação toma como perspectiva o fato de que as peças centrais de cada face do cubo não se deslocam.

As faces do cubo serão chamadas de U, L, F, R, B e D (*up, left, front, right, back e down*, respectivamente), de acordo com a notação adotada pela comunidade envolvida com o cubo mágico.

4 Implementação

A implementação do trabalho se divide em duas partes: o algoritmo genético e o cubo mágico em si. O trabalho foi implementado integralmente em Python 3.5. A única biblioteca não-pertencente à

biblioteca padrão da linguagem utilizada foi NumPy[2], que permite representar vetores e matrizes de maneira eficiente em memória e dispõe de algumas funções matemáticas práticas.

Foi tratado neste trabalho apenas o cubo de lado de tamanho 3, devido à complexidade adicional de tratar cubos maiores.

Para fins de evitar um nome completamente genérico para eventual publicação deste trabalho na internet ou outros meios, a implementação do algoritmo genético foi chamada de **Rubicon** (em homenagem ao rio de mesmo nome e devido à similaridade entre a palavra e o nome do criador do cubo). O nome não é utilizado nesta documentação, mas é o nome do pacote da implementação Python e é utilizado ao longo do código. Por isso e para evitar confusão, ele é mencionado nesta seção brevemente.

4.1 Cubo mágico

São várias as representações possíveis para o cubo mágico. Ele pode ser representado como uma matriz tridimensional 3x3x3 em que cada elemento é um *cubelet*, como uma lista de seis lados que são matrizes 3x3 de *facelets* ou cores, ou mesmo como uma permutação dos seus 54 *facelets*. A numeração dos facelets e a coloração das faces utilizada no cubo resolvido pode ser vista na tabela 1.

Tabela 1: Cubo Mágico com *facelets* numerados

			0	1	2						
			3	4	5						
			6	7	8						
9	10	11	18	19	20	27	28	29	36	37	38
12	13	14	21	22	23	30	31	32	39	40	41
15	16	17	24	25	26	33	34	35	42	43	44
			45	46	47						
			48	49	50						
			51	52	53						

Nesta implementação, foram utilizadas em ocasiões diferentes a representação de seis faces 3x3 e a representação da permutação de 54 *facelets*. Me referirei a elas daqui em diante como representação tridimensional e representação planificada, respectivamente. *(Portanto, representação tridimensional NÃO se refere à matriz 3x3x3 de cubelets. A primeira dimensão da matriz corresponde à face do cubo.)*

As representações foram mescladas porque cada uma tem suas vantagens. A representação tridimensional é útil para realizar movimentos no cubo, pois rotações são muito mais naturais nessa forma. Por outro lado, a representação planificada se mostra bem mais eficiente e prática para algumas operações, devido à forma como matrizes NumPy funcionam.

4.1.1 Movimentos

Movimentos para o cubo convencional geralmente são representados de acordo com uma notação que utiliza dezoito movimentos diferentes - três por face, referentes à rotação em sentido horário, anti-horário e de 180 graus. Para a face F, esses movimentos são F, F' e F2, respectivamente. Para fins de implementação, porém, é suficiente implementar o movimento em sentido horário e repetí-lo duas ou três vezes para executar os outros dois.

A implementação dos movimentos se resume, portanto, a uma única função *rotate*, que recebe como parâmetro o identificador da face e o número de vezes que será realizado. Essa função realiza duas operações principais: rotacionar os *facelets* da face em si e girar os facelets das faces adjacentes. A primeira parte é uma simples operação em matrizes que, inclusive, NumPy já implementa por meio da função `rot90`.

A rotação dos quadrados das faces adjacentes é mais trabalhosa. O arquivo-fonte *rubikscube.py* contém uma função `adjacent_vector_indices`, que retorna uma lista de identificadores de vetores (linhas ou colunas) adjacentes a uma face, em sentido horário. Esses identificadores são definidos em três campos: face, linha e coluna. Os campos de linha e coluna podem ser definidos como números inteiros ou sentidos, em que, para definir um vetor, sempre serão um inteiro e um sentido, necessariamente. Sentido é um campo binário que pode ser positivo (cima para baixo/esquerda para direita) ou negativo (vice-versa).

Como exemplo, os vetores adjacentes da face **esquerda** do cubo são, em sentido horário (face, vetor, sentido):

1. U, coluna 0, positivo;
2. F, coluna 0, positivo;
3. D, coluna 0, positivo;
4. B, linha 0, negativo.

Sabendo esses quatro vetores, rotacionar o anel de vetores adjacente a uma face consiste de apenas deslocá-los em sentido horário. (Atribuir o primeiro para o segundo, o segundo para o terceiro etc.)

4.1.2 Compilação dos movimentos

A implementação descrita anteriormente é relativamente prática, mas, por ser realizada iterativamente no cubo tridimensional, seu desempenho não é bom. No entanto, o movimento realizado independe da configuração do cubo, e consiste apenas em realizar algumas trocas de posição de *facelets*. Para otimizar o desempenho das rotações, - e, consequentemente, do GA, visto que essa seria a operação mais custosa no cálculo da fitness a cada geração, - foi implementado um "compilador" de movimentos, por meio da função `freeze_move` do arquivo *movement.py*.

Essa função realiza um movimento em cubo tridimensional e compara o cubo após o movimento com o cubo antes da movimentação. Essa comparação permite a obtenção dos índices que mudaram. Com esses índices, é gerada uma nova função bem mais eficiente que realiza a mesma operação de movimento sobre o cubo planificado, apenas trocando índices de lugar. Esse processo é realizado para cada uma das dezoito configurações válidas da função de rotação.

4.2 Algoritmo genético

O algoritmo genético (GA) foi desenvolvido de forma genérica, inicialmente, na pasta *ga* do código fonte do trabalho. Essa pasta corresponde a uma biblioteca de execução de um GA genérico, independente do problema do Cubo Mágico. O algoritmo desenvolvido segue a forma de um algoritmo genético convencional:

1. Inicializa-se a população (fora do algoritmo principal, permitindo, assim, que seja inicializada a população com um indivíduo não-aleatório, caso necessário);
2. Tem início uma geração;
3. É calculada a fitness de todos os indivíduos por meio de uma função de fitness;
4. São selecionados filhos com base nessa fitness por meio de um operador de seleção (com elitismo);
5. A população é variada por meio de uma função de variação da população;
6. Dados sobre a fitness e a população são registrados em um dicionário de estatísticas;
7. Os filhos se tornam a nova população e começa outra geração.

Note que os operadores de seleção, variação e o cálculo de fitness não são definidos no algoritmo genético. Ao invés disso, o GA recebe como parâmetro um objeto do tipo *Toolkit* que contém esses operadores como métodos estáticos.

4.2.1 Toolkit

O Toolkit é uma classe definida na biblioteca do GA desta implementação que não possui métodos implementados, mas requer a implementação de pelo menos quatro métodos: `create`, `select`, `vary` e `fitness`, relativos à criação de um novo indivíduo, à seleção de uma nova população, à variação de uma população e do cálculo da fitness de um indivíduo.

Esses métodos não recebem parâmetros adicionais. Isso pode parecer um problema, mas na verdade é uma maneira elegante de simplificar a interface dessas funções sem perder a flexibilidade que parâmetros proporcionam através do poder de Python como uma linguagem funcional. Se uma função recebe parâmetros, ela pode ser transformada em uma versão com os parâmetros já definidos através de `functools.partial`. Dessa forma, na inicialização do *Toolkit*, são definidas versões parciais dos operadores, tornando o código mais legível no uso subsequente dos mesmos.

4.2.2 Operadores genéticos genéricos

Alguns dos operadores utilizados não são específicos para o problema em questão, e estão disponíveis na biblioteca do GA:

- `vary`: recebe como parâmetro uma população e varia a mesma por meio de operadores `mate` e `mutate`, correspondentes ao cruzamento e à mutação, respectivamente;
- `sel_tourn`: realiza seleção por torneio de uma população, retornando uma lista de filhos;
- `sel_best`: seleciona os n melhores indivíduos de uma população; (Utilizado para implementar elitismo no algoritmo)
- `size_limit`: decora outro operador, limitando o tamanho máximo do indivíduo a ser obtido. Caso o indivíduo gerado seja maior que o limite, o operador retorna o indivíduo original.

Os operadores em questão recebem vários parâmetros que controlam o comportamento dos mesmos, como o tamanho do torneio e as probabilidades de cruzamento/mutação. Para entender melhor tais detalhes de implementação, é recomendável que seja lido o código-fonte do programa, que contém diversos comentários sobre essas características.

4.3 Aplicação do GA ao problema

4.3.1 Indivíduos

Os indivíduos do algoritmo genético para o cubo mágico são sequências de movimentos dentre os dezoito possíveis. Internamente, os indivíduos são representados como vetores de inteiros de 0 a 17, que se mapeiam a cada um dos movimentos. A inicialização dos indivíduos é aleatória, com tamanhos mínimo e máximo iniciais definidos na configuração de execução.

Vale notar que, como os indivíduos determinam um conjunto de instruções e como existem múltiplos genótipos capazes de se mapear ao mesmo fenótipo, o GA utilizado na verdade é um algoritmo de Programação Genética (GP). Isso acarreta, também, em vários dos problemas notórios em GP, como **bloat**, que acontece com frequência quando ocorre uma sequência redundante de movimentos. (e.g.: F F F F, U2 U2, B B' etc.)

4.3.2 Operadores genéticos

O arquivo-fonte *rubicon_toolkit.py* contém os operadores genéticos específicos ao problema do Cubo Mágico. Eles são:

- `cx_point`: realiza o cruzamento de ponto único entre dois indivíduos;
- `mutate_replace`: realiza a mutação de um indivíduo, removendo um trecho contíguo e substituindo o mesmo por outro trecho, não necessariamente do mesmo tamanho;

O cruzamento de ponto único foi escolhido porque é uma forma simples e eficaz de cruzamento. Ele muitas vezes é desconsiderado porque ele permite que uma parte ruim do genótipo se preserve no meio de partes boas e dificilmente seja removida. No entanto, para o problema do cubo mágico, outro movimento inserido em uma posição anterior ou posterior pode cancelar o aspecto "ruim" de tal gene. Por isso, esse problema não se manifesta em peso.

O arquivo também contém a função de criação de novo indivíduo previamente descrita, assim como a especialização da classe *Toolkit*, *RubiconToolkit*, que instancia funções parciais para cada operador com os parâmetros definidos nas configurações.

A função de fitness escolhida, por sua vez, leva em consideração o fato de que avaliar o quão próximo de uma solução o cubo mágico está não é evidente com base em uma única métrica visual. Dessa forma, são combinados na mesma função de fitness quatro aspectos diferentes: *facelets* de cor correta, *cubelets* na posição correta, distância de cada *facelet* à posição correta e tamanho do indivíduo.

4.3.3 *Facelets* e *cubelets*

As *facelets* de cor incorreta e *cubelets* na posição incorreta (a orientação dos mesmos não importa) são funções de fitness que determinam a qualidade do posicionamento independente das peças do cubo. O problema desse tipo de fitness é que o cubo não funciona de maneira independente: as peças se movimentam em conjunto, e aí mora a complexidade de resolvê-lo.

No entanto, ao combinar ambas métricas, a probabilidade de que um cubo tenha muitas *facelets* e *cubelets* na posição correta simultaneamente tem maior correlação com a proximidade do cubo à solução real. Essa métrica de avaliação existe na literatura e apresenta bons resultados na aplicação como fitness de algoritmo genético para resolução do cubo[4].

4.3.4 Distância gráfica de solução

Este trabalho propõe uma métrica adicional para avaliação da solução do cubo mágico. Essa métrica avalia a proximidade de um *facelet* no cubo à sua posição no cubo resolvido. Para isso, é importante notar que, enquanto a métrica anterior relativa aos facelets trata todos os *facelets* de mesma cor como iguais, isso não necessariamente é a melhor forma de tratá-los. No cubo 3x3x3, todo *facelet* de uma certa cor está associado a um conjunto distinto de outros *facelets* no seu *cubelet*. Isso significa que cada *facelet* é **único** no cubo mágico. (Isso não é verdade para cubos de dimensões maiores. Por isso foi impossível generalizar esse modelo para outros tamanhos.)

Sabendo disso, é possível determinar o número mínimo de movimentos no cubo que são necessários para levar um *facelet* à sua posição inicial. Essa distância pode ser calculada por meio de uma modelagem dos movimentos do cubo em um grafo.

Seja $G(V, E)$ um grafo simples com 54 vértices, cada um correspondente a um *facelet* do cubo. Existe uma aresta entre os vértices u e v se e somente se é possível, por meio de algum dos 18 movimentos definidos previamente, movimentar o facelet da posição u para a posição do facelet v. A distância entre um facelet f para todos os outros facelets pode ser, então, calculada através da árvore de busca em profundidade a partir do vértice f. Caso não seja possível levar um facelet de uma posição u para a posição j, a distância entre os dois é infinita.

Realizando essa busca em profundidade a partir de todos os vértices, é possível obter uma matriz n por n (onde n é o número de vértices de G, 54) $D_{n\times n}$. $D_{i,j}$ é a distância mínima entre os facelets i e j.

Seja C_i o cubo mágico de entrada do problema após a aplicação dos movimentos correspondentes ao indivíduo i. A distância de i é a soma, para cada *facelet* $f \in C$, da distância mínima entre esse facelet e a sua posição no cubo resolvido. Ou seja:

$$\text{solDist(C)} = \sum_{i=0}^{n} D_{i,C(i)}, \tag{2}$$

onde $C(i)$ é o *facelet* na i-ésima posição do cubo.

4.3.5 Fitness combinada

A combinação das funções de fitness foi feita de maneira a dar pesos equivalentes - isso é, normalizar as escalas de cada uma das três funções de fitness. As escalas de cada fitness estão a seguir:

1. Facelets de cor errada (*wrongColor)*: de 0 a 48 (seis facelets centrais não se movem);
2. Cubelets na posição errada (*wrongCubelets*): de 0 a 20 (seis cubelets centrais não se movem);
3. Distância gráfica (*graphDistance*): de 0 a 96 (cada facelet pode estar, no máximo, a dois movimentos da sua posição correta, exceto pelos centrais).

Além disso, foi adicionado à função de fitness um termo adicional correspondente ao logaritmo do tamanho dos indivíduos (*len*) dividido por uma constante relativamente grande (10) foi escolhido. Isso se deve à intenção de que os indivíduos sejam os menores possíveis, mas à baixa prioridade desse critério em relação à corretude da solução. A fitness final é:

$$\frac{wrongColor(C_i)}{2.4} + wrongCubelets(C_i) + \frac{graphDistance(C_i)}{4.8} + \frac{\log_{10} len(ind)}{10} \tag{3}$$

4.4 Estatísticas e gráficos gerados

Cada conjunto de execuções do algoritmo genético gera uma pasta que contém todas as informações pedidas na especificação do trabalho sobre o mesmo. São mantidas, para cada execução:

- A fitness mínima, máxima, média e o desvio padrão das fitnesses por geração;
- O tamanho mínimo, máximo, médio e o desvio padrão dos tamanhos dos indivíduos por geração;
- O número de indivíduos que melhorou desde a geração anterior, por geração;
- O número de indivíduos repetidos na população, por geração;
- O melhor indivíduo final e sua fitness;
- O cubo mágico final do melhor indivíduo;
- O tempo total de execução do GA;
- Todos os indivíduos da população final, salvos em um formato *pickle*, que permite que eles sejam carregados novamente para um programa com facilidade, caso necessário.

Quando executadas múltiplas execuções em conjunto, o programa também gera um sumário do conjunto de execuções, que gera estatísticas adicionais sobre as estatísticas de cada execução. São gerados para com os valores de cada execução:

- Para cada geração:
  - Fitness e tamanho dos indivíduos:
    - O mínimo, a média e o desvio padrão dos valores mínimos;
    - O mínimo, a média e o desvio padrão das médias;
    - A média dos valores máximos;
    - A média dos desvios padrão.
  - O mínimo, o máximo, a média e o desvio padrão do número de indivíduos que melhorou;
  - O mínimo, o máximo, a média e o desvio padrão do número de indivíduos repetidos na população;
- Os melhores indivíduos de cada execução e suas fitnesses;
- O melhor de todos os indivíduos, sua fitness e o cubo mágico gerado por sua solução.

5 Execução

Para executar o programa, é necessária uma instalação de Python 3 (o programa foi testado em CPython 3.5.2) com o módulo NumPy. Adicionalmente, para gerar os gráficos das estatísticas por geração, é necessário ter instalado o módulo Matplotlib e, opcionalmente, o módulo Seaborn (caso esse último falte, os gráficos utilizarão um estilo de cores diferente do presente neste relatório, mas ainda serão gerados).

A execução consiste em chamar o interpretador Python para o programa, fornecendo um parâmetro adicional correspondente ao arquivo de configuração da execução. Esse arquivo é um JSON de formato específico e vários exemplos estão presentes na pasta `config/`.

Exemplo de execução:

```
python3 rubicon/ config/run.json
```

Onde `rubicon/` corresponde à pasta do pacote principal do programa (Python, quando fornecido uma pasta, busca por um arquivo `__main__.py` para executar, que é o arquivo principal do programa desenvolvido neste trabalho.

6 Experimentos

Após a implementação deste trabalho, foram realizados diversos experimentos. Cada experimento consiste em um conjunto de execuções do algoritmo genético com um determinado conjunto de parâmetros de execução. Para facilitar a variação dos mesmos, o programa implementado utiliza um arquivo de configuração, no qual podem ser especificados todos os parâmetros a ser variados nos experimentos a seguir. No início da experimentação, esses parâmetros foram inicializados com valores escolhidos de maneira arbitrária, mas intuitiva. A tabela 2 contém a lista dos parâmetros, uma breve descrição e seus valores iniciais.

Tabela 2: Parâmetros de execução

Nome	Descrição	Valor inicial
InitMinSize	Tamanho inicial mínimo dos indivíduos	5
InitMaxSize	Tamanho inicial máximo dos indivíduos	15
MutMinSize	Tamanho mínimo dos trechos adicionados ou removidos por mutação	1
MutMaxSize	Tamanho mínimo dos trechos adicionados ou removidos por mutação	10
IndMaxSize	Tamanho máximo dos indivíduos em qualquer ocasião	100
PopSize	Número de indivíduos da população	100
Gens	Número de gerações do algoritmo genético	100
TournSize	Tamanho do torneio para seleção	3
NumElitism	Número de melhores indivíduos reproduzidos por elitismo	1
CxProb	Probabilidade de cruzamento	0.9
MutProb	Probabilidade de mutação	0.1
Runs	Número de execuções do algoritmo genético por experimento	30

A variação dos parâmetros ao longo dos experimentos é feita sempre de maneira iterativa - apenas um parâmetro é modificado de um experimento para o outro. Nas seções a seguir, serão denotadas apenas as mudanças realizadas nas configurações - não serão listadas todas elas novamente. Além disso, serão comentadas apenas as informações que foram importantes para a tomada de decisão nos experimentos. Caso seja de interesse, todas as informações mencionadas na seção de estatísticas e todos os gráficos estão disponíveis na pasta `runs`, onde há subdiretórios que contém cada experimento.

6.1 Número de indivíduos

O primeiro momento da experimentação é dedicado à definição de dois parâmetros bastante importantes: o número de indivíduos e o número de gerações do algoritmo genético.

No experimento 1, é variado o número de indivíduos de cada geração. São testados quatro valores: 50 (figura 1), 100 (figura 2), 200 (figura 3) e 400 (figura 4). Avaliando a convergência dos gráficos de fitness e os valores da média das fitnesses mínimas entre os quatro conjuntos de execuções, é possível perceber que não há ganho significativo de desempenho do algoritmo genético após 100 indivíduos. Por isso, nos experimentos seguintes, *PopSize* é definido como 100.

6.2 Número de gerações

No segundo experimento, é alterado o número de gerações para 800 e o algoritmo genético é executado. A intenção desse experimento era determinar em que geração havia convergência da fitness, visto que os gráficos das fitnesses das execuções prévias (figura 2, por exemplo) não convergiam por completo. Similarmente às execuções anteriores, nota-se um declive acentuado na fitness nas primeiras gerações, mas não há convergência total dos indivíduos: continua havendo decaimento lento da fitness.

Há queda significativa da média das fitnesses mínimas e do valor mínimo absoluto, como se pode ver no gráfico sumarizado das fitnesses, na figura 5. A média das fitnesses médias também é reduzida. O tempo de execução das instâncias com 800 gerações não é inviável, e houve melhorias de peso. Portanto, foi mantido o valor de 800 gerações, a ser revisto em experimentos posteriores.

6.3 Coeficiente do tamanho dos indivíduos na fitness

Como mencionado na introdução, o *God's Algorithm* revela que é necessário no máximo 20 passos para resolver o cubo mágico, a depender da instância. No entanto, sequências de passos geradas por um algoritmo estocástico dificilmente são tão eficientes e, por isso, é de se esperar que um número razoável de passos seja necessário para resolver grande parte das instâncias.

Nas execuções anteriores, é possível perceber que o tamanho dos indivíduos é menor que o desejado. A média de tamanho dos indivíduos no experimento 2 se mantém relativamente pequena, embora crescente, ao longo das gerações, como se pode ver na figura 6.

A redução do coeficiente do tamanho dos indivíduos na função de fitness de 0.1 para 0.01 não trouxe o resultado esperado, porém: o gráfico das novas execuções (figura 7 revela que o problema persiste. Isso indica que o termo de tamanho dos indivíduos na fitness pode não ser o fator que limita o tamanho dos indivíduos. A redução do coeficiente foi mantida, visto que aumentá-la novamente não traria benefícios.

6.4 Aumento dos tamanhos de inicialização

Outra forma de tentar tratar o problema do tamanho insatisfatório dos indivíduos é através do tamanho de inicialização dos mesmos. A motivação para essa mudança foi a evolução do tamanho dos indivíduos em execuções distintas. Enquanto, em grande parte das execuções, o tamanho convergia rapidamente para um valor pequeno, (figura 8) ocorriam execuções em que o contrário era realidade: o tamanho dos indivíduos estabilizava em um valor alto. (Figura 9.) Algumas vezes, inclusive, eles aparentemente só foram limitados pelo limite estático de tamanho 100 imposto pelos parâmetros do problema.

O aumento dos tamanhos mínimo e máximo de inicialização foi uma forma de tentar estimular a divergência do tamanho dos indivíduos. Isso pode se mostrar problemático em alguns contextos, devido à ocorrência de bloat, mas nessa situação havia preocupações distintas. Infelizmente, isso não foi suficiente: o gráfico sumarizado dos tamanhos dos indivíduos (figura 10) revela que o aumento nos tamanhos iniciais foi completamente infrutífero, com a convergência imediata do tamanho dos mesmos.

Figura 1: Fitness de 50 indivíduos (todas as execuções)

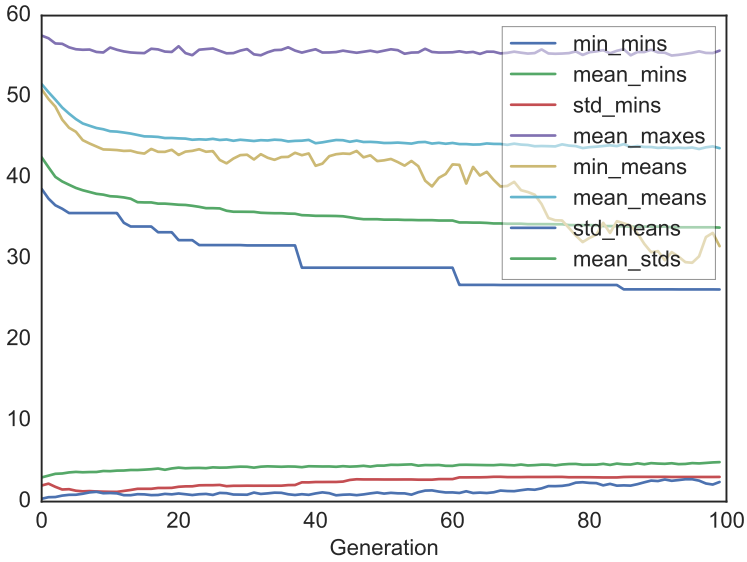


Figura 2: Fitness de 100 indivíduos (todas as execuções)

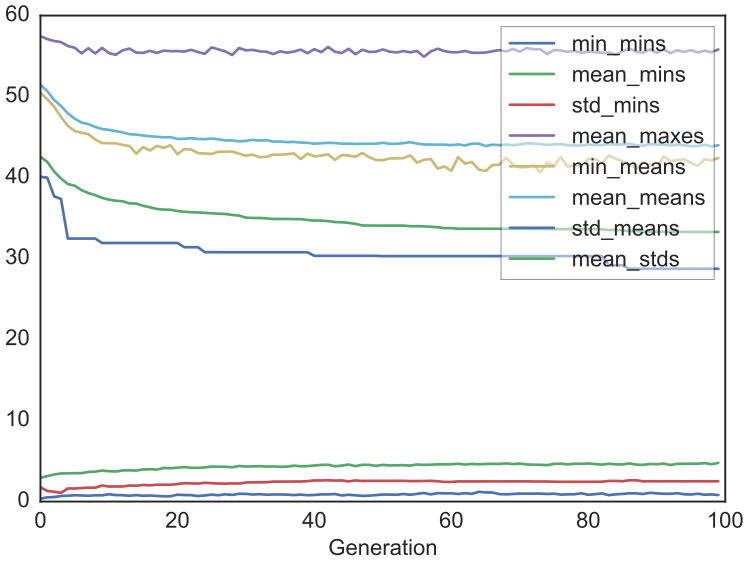


Figura 3: Fitness de 200 indivíduos (todas as execuções)

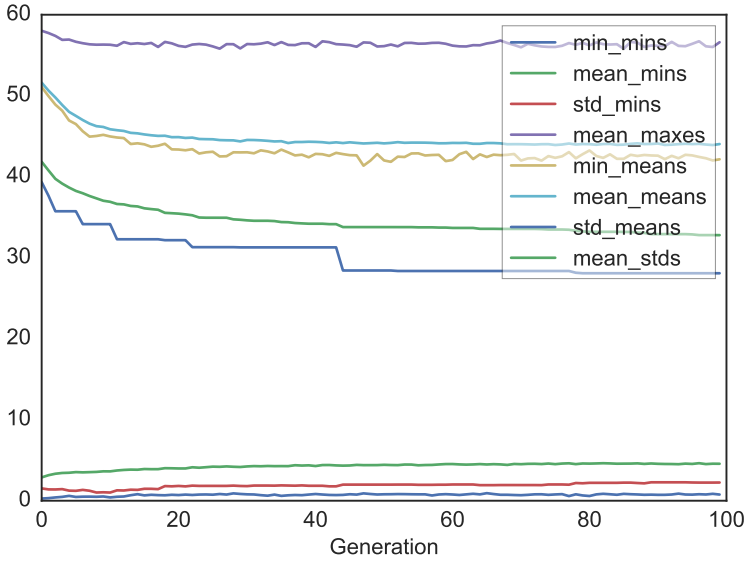


Figura 4: Fitness de 400 indivíduos (todas as execuções)

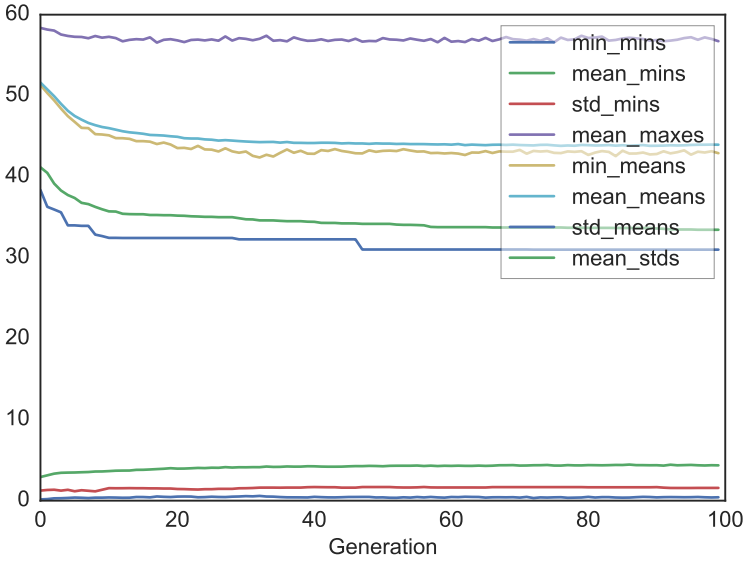
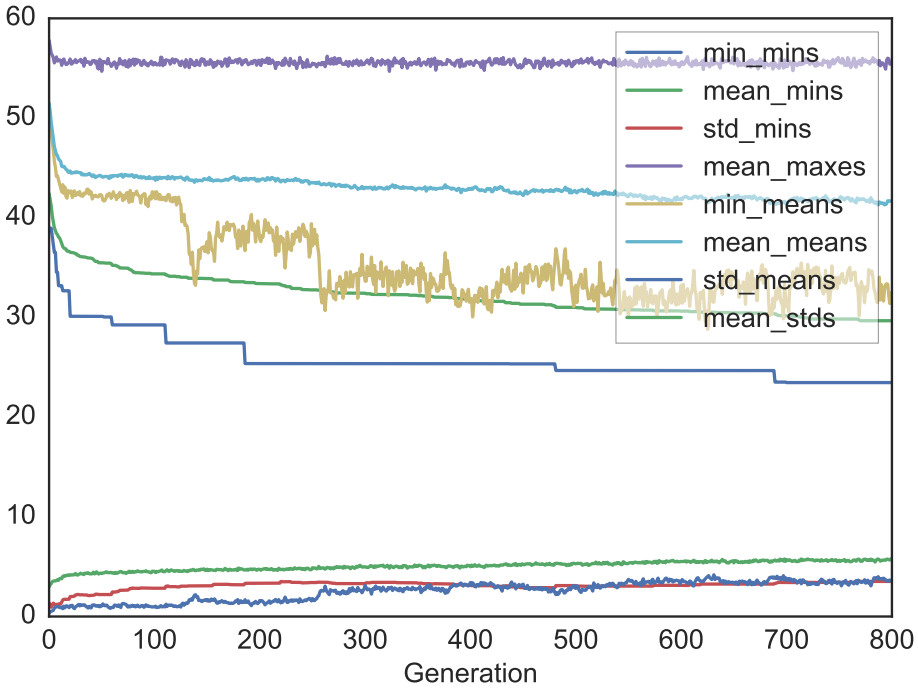


Figura 5: Fitness com 800 gerações



Novamente, a modificação nos parâmetros foi mantida, já que não há por que reduzir os parâmetros de volta aos valores originais.

6.5 Redução do tamanho do torneio

Voltando à questão da fitness, a quase-convergência rápida e brusca da fitness da população em quase todas as execuções poderia ser razão da pressão seletiva do algoritmo. Embora o tamanho do torneio já não fosse particularmente grande (no caso, 3), reduzí-lo é uma forma garantida de reduzir a pressão seletiva e, consequentemente, analizar o impacto da mesma nas populações. Dessa forma, foi realizada um novo experimento com torneios de tamanho 2.

Os resultados foram bastante insatisfatórios: houve piora tanto na média das fitnesses mínimas quanto na média das fitnesses médias. Evidentemente, a pressão seletiva não só é suficiente como pode ser uma maneira promissora de melhorar o desempenho do sistema. A mudança não foi mantida.

6.6 Aumento do tamanho do torneio

Após o experimento anterior, é intuitivo tentar aumentar ainda mais o tamanho do torneio para avaliar os impactos da mudança no sentido oposto, visto que diminuí-lo piorou o desempenho. O aumento do torneio para tamanho 5 não trouxe melhoria significativa em relação ao torneio de tamanho 3: os resultados foram quase que completamente equivalentes. O valor foi mantido daqui em diante para facilitar a experimentação iterativa, por não haver diferença significativa nos resultados.

6.7 Aumento da mutação

Como sugerido na especificação do trabalho, este experimento consiste em testar uma configuração diferente de proporcionalidade entre os operadores de variação. As execuções anteriores utilizavam crossover alto e mutação baixa. Neste conjunto de execuções, a taxa de mutação foi aumentada para 0,3 e a taxa de cruzamento foi reduzida para 0,6.

Figura 6: Tamanho dos indivíduos antes da modificação (todas as execuções)

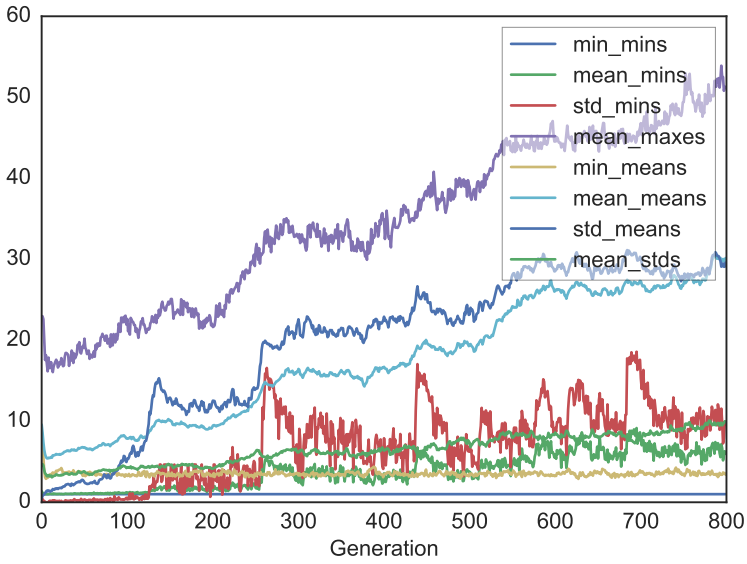
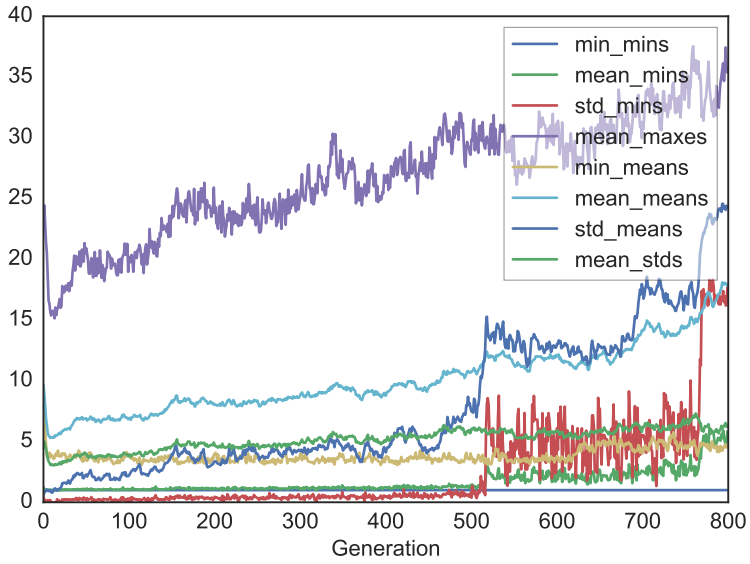


Figura 7: Tamanho dos indivíduos depois da modificação (todas as execuções)



Os resultados obtidos (figura 12) foram piores do que o obtido anteriormente (figura 11) por uma margem bem significativa, no que cabe à fitness mínima média e à média das fitnesses médias de todas as execuções. O melhor indivíduo geral também piorou, o que significa que o ganho de diversidade da taxa de mutação aumentada não compensou os potenciais efeitos destrutivos da mesma e a redução nos benefícios do cruzamento.

No entanto, não é possível perceber diferença significativa entre a taxa de melhoria dos indivíduos por geração entre as execuções com mais cruzamento (figura 13) e as com maior taxa de mutação (figura 14). Como não houve melhorias de grande significância desde o princípio da execução dos experimentos, faz sentido interpretar que nenhum dos dois operadores está causando grande impacto na população.

A taxa de melhoria por geração fica estável em pouco abaixo de 50% em ambos conjuntos de execuções. Isso significa que existe um equilíbrio entre a melhoria e piora dos indivíduos, e mostra a estagnação rápida do método. Essa taxa de melhoria cai rapidamente, similarmente à queda brusca que ocorre com a fitness nas primeiras gerações.

6.8 Aumento no número de gerações

A não-convergência da curva de média das fitnesses mínimas é incômoda, e, para avaliar o impacto dessa métrica no melhor indivíduo ao fim das execuções, fez sentido, dado que há tempo, executar instâncias do algoritmo genético com um número muito aumentado de gerações. Dessa forma, foi alterado o número de gerações de 800 para 7000.

Houve, assim como no primeiro aumento do número de gerações, uma queda significativa na média das fitnesses mínimas por geração e na média das fitnesses médias também. Além disso, o melhor indivíduo obtido foi realmente melhor que na execução anterior. Contudo, ele não foi o melhor indivíduo obtido ao longo de todas as execuções, o que indica que as melhorias obtidas se devem principalmente ao acaso e ao efeito descontrolado dos operadores genéticos. Finalmente, é difícil acreditar que aumentar mais o número de gerações vá resolver o problema, ao menos se a intenção é fazê-lo de forma guiada: é perfeitamente crível que os resultados venham sendo, a partir do primeiro momento de convergência da população, decorrentes de fatores aleatórios.

O número de gerações foi mantido para o próximo experimento.

Figura 8: Tamanho convergente dos indivíduos

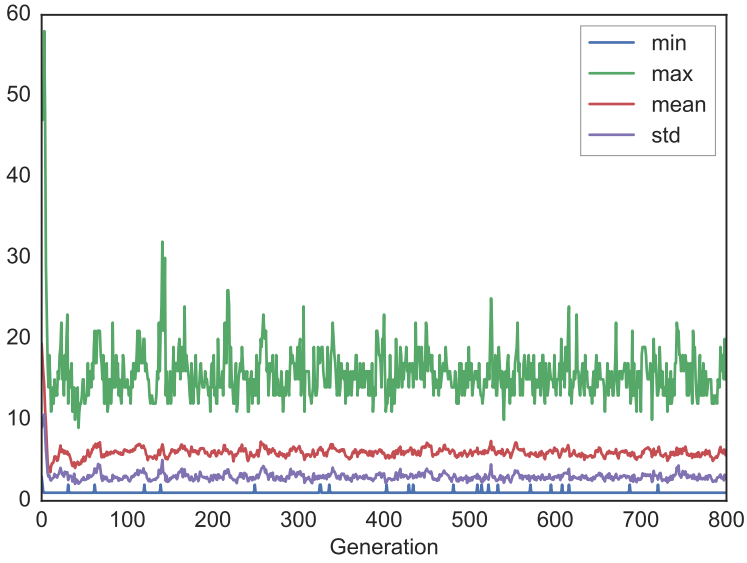
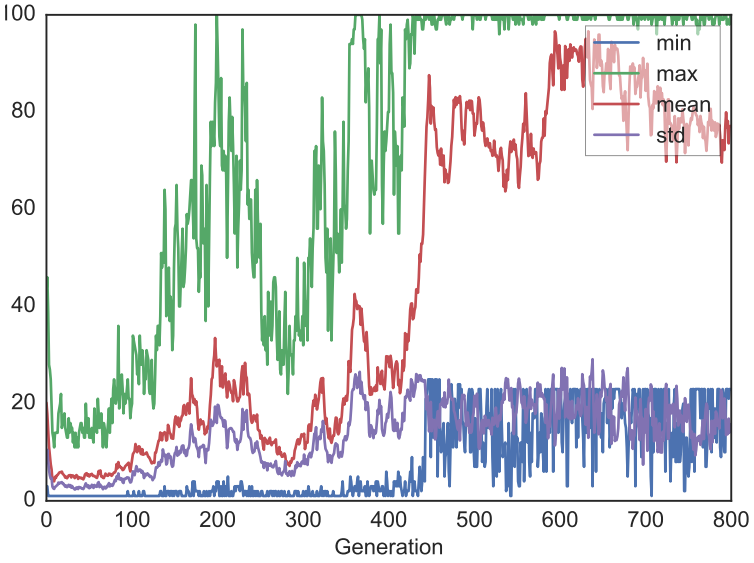


Figura 9: Tamanho divergente dos indivíduos



6.8.1 Aumento no número de indivíduos

Outro teste que, intuitivamente, deveria causar apenas efeitos positivos sobre o sistema é o aumento da população utilizada. Ela foi triplicada, portanto (100 para 300 indivíduos), e as execuções foram realizadas.

A fitness do melhor indivíduo nesse conjunto de execuções foi melhor, mas por pouco, equiparada à melhor fitness encontrada até então. A média das fitnesses mínimas foi muito similar, porém, o que leva mais uma vez à intuição de que isso ocorreu por sorte. O algoritmo genético não parece estar funcionando de acordo com o esperado.

6.8.2 Remoção do elitismo

O penúltimo experimento realizado envolve, como previsto na especificação do trabalho, testar o desempenho do algoritmo genético sem elitismo. A execução foi feita com tamanho da população e número de gerações de volta aos valores antes dos aumentos realizados nos dois experimentos anteriores - ou seja, 800 gerações e 100 indivíduos, como no experimento da seção 6.6.

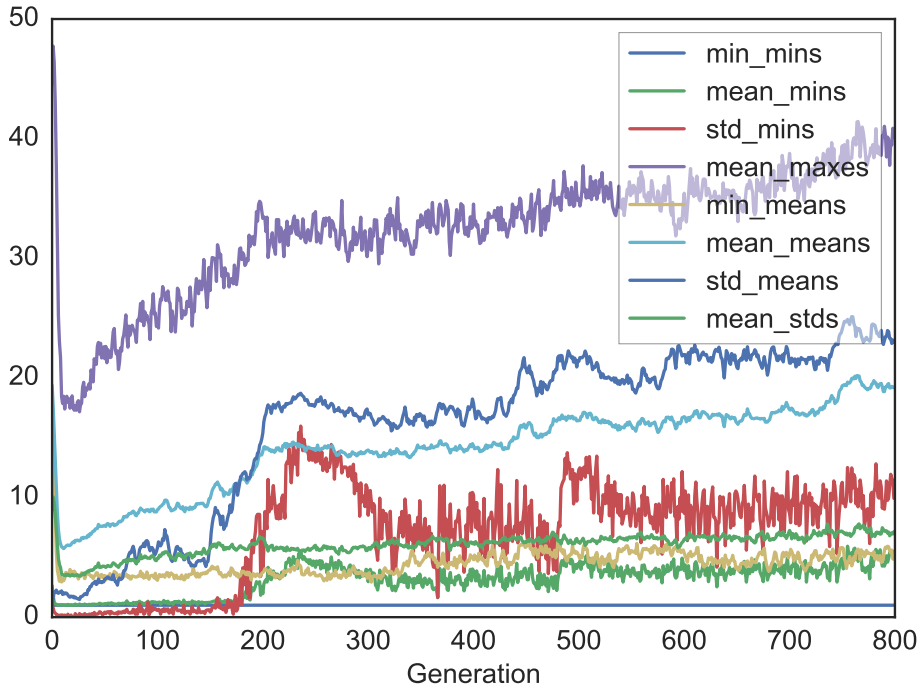
Os resultados foram, como era de esperar, *terríveis*. A média das fitnesses mínimas e médias subiu muito significativamente, a fitness mínima geral subiu diversas vezes nas primeiras gerações e o algoritmo convergiu para valores muito piores do que mesmo o que foi obtido nos primeiros experimentos, como pode ser visto na figura 15.

6.8.3 Outras instâncias do problema

Os experimentos realizados previamente foram todos feitos com base na instância de teste "*in1*", providenciada juntamente à especificação do trabalho. Havia duas outras instâncias "*in2*" e "*in3*", que foram testadas nesse experimento adicional. Os parâmetros utilizados foram os mesmos do experimento da seção 6.6.

Os gráficos de fitness obtidos ao final dessas execuções (figuras 16 e 17) se assemelham ao gráfico da fitness do experimento de mesmos parâmetros realizado para o cubo *in1*. Vale notar que o cubo *in3* obteve resultados consideravelmente piores. Sua fitness média inicial é maior, o que pode significar que ele se trata de uma instância mais distante da solução e, consequentemente, mais difícil que *in2* ou *in1*.

Figura 10: Gráfico sumarizado de tamanho após a modificação



7 Melhores indivíduos por instância

Nesta seção, são expostos os melhores indivíduos encontrados para cada instância de teste do cubo de lado 3 (*in1*, *in2* e *in3*), ao longo de todas as execuções realizadas para cada um. Vale notar que os parâmetros das execuções que encontraram cada um desses indivíduos é diferente; eles não provêm de experimentos consistentemente realizados e avaliados.

7.1 in1

Movimentos: U R2 F R2 F2 F R2 F F R F F R2 F R2 F R2 F F R2 F R2 F F R F R2 F R2 F R F2 R2 F2 R F R2 F F F R2 F2 R2 F F2 R2 F F R F R2 F R2 F R F2 R2 F2 R F R2 F R2 F F F U R2 U R2 U R2 D2 R D2

Fitness: 15,3939162705

Tabela 3: Melhor indivíduo da entrada in1

			0	1	2						
			3	4	5						
			6	7	8						
9	10	11	18	19	20	27	28	29	36	37	38
12	13	14	21	22	23	30	31	32	39	40	41
15	16	17	24	25	26	33	34	35	42	43	44
			45	46	47						
			48	49	50						
			51	52	53						

Figura 11: Fitness antes da modificação

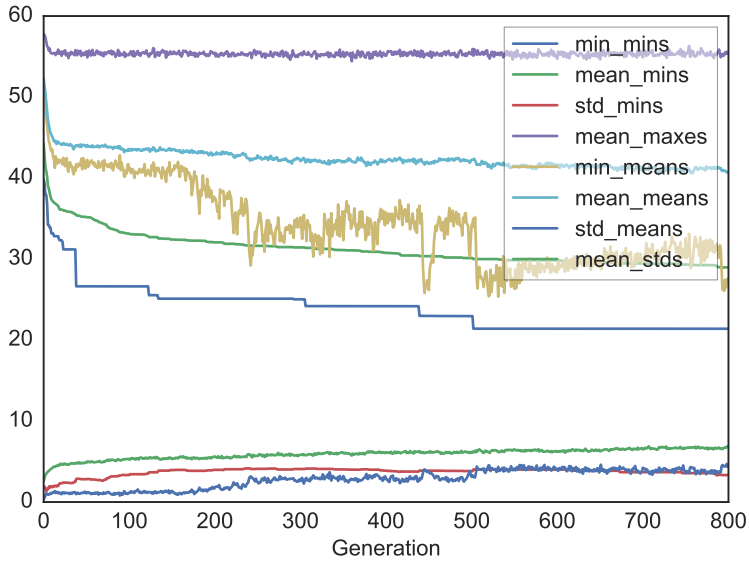
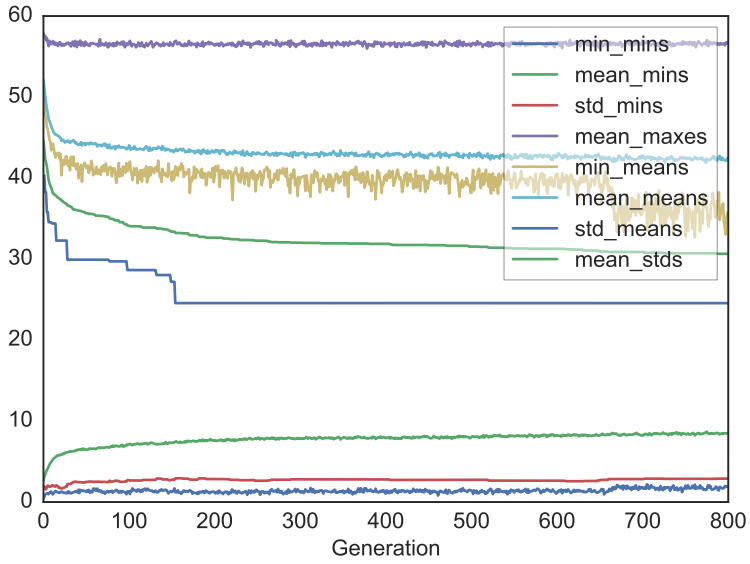


Figura 12: Fitness após a modificação



7.2 in2

Movimentos: D' R F D' R' R' F F R' R' U D
Fitness: 20,4994968883

Tabela 4: Melhor indivíduo da entrada in2

			0	1	2						
			3	4	5						
			6	7	8						
9	10	11	18	19	20	27	28	29	36	37	38
12	13	14	21	22	23	30	31	32	39	40	41
15	16	17	24	25	26	33	34	35	42	43	44
			45	46	47						
			48	49	50						
			51	52	53						

7.3 in3

Movimentos: F' U U F2 U U U F2 U U F2 U U F2 U F2 U U U F2 U F2 U U U F2 U F2 U U U F2 U U F2 U U F' R D2 R2 U U F2 U R R U2 D2 R2 D'
Fitness: 23,9643941878

8 Sobre a qualidade dos resultados

Evidentemente, o algoritmo genético falhou em seu propósito - mesmo as melhores instâncias obtidas são, ainda, consideravelmente distantes da solução ótima. Várias são as razões pelas quais isso pode ter ocorrido. Uma das mais prováveis é a função de fitness, que mede aspectos independentes do cubo, mas possivelmente falha em avaliá-lo como um todo, apesar de todo o esforço feito a respeito disso.

Figura 13: Melhorias por geração antes da modificação

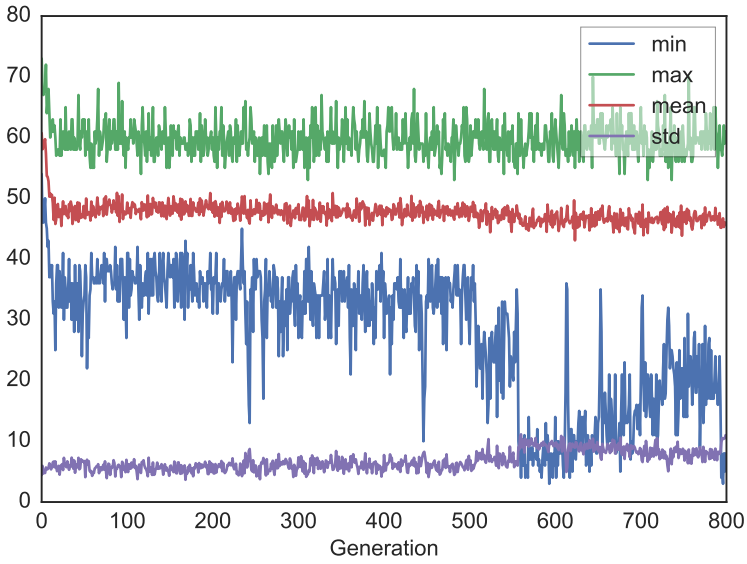


Figura 14: Melhorias por geração após a modificação

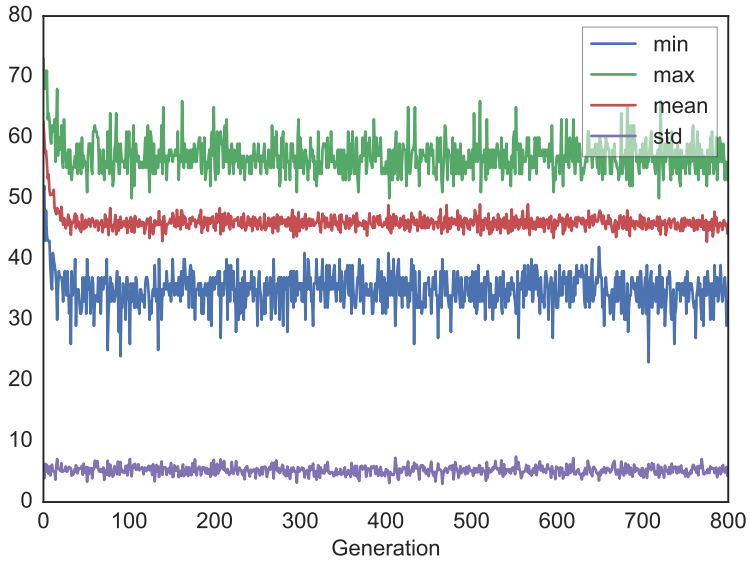


Tabela 5: Melhor indivíduo da entrada in3

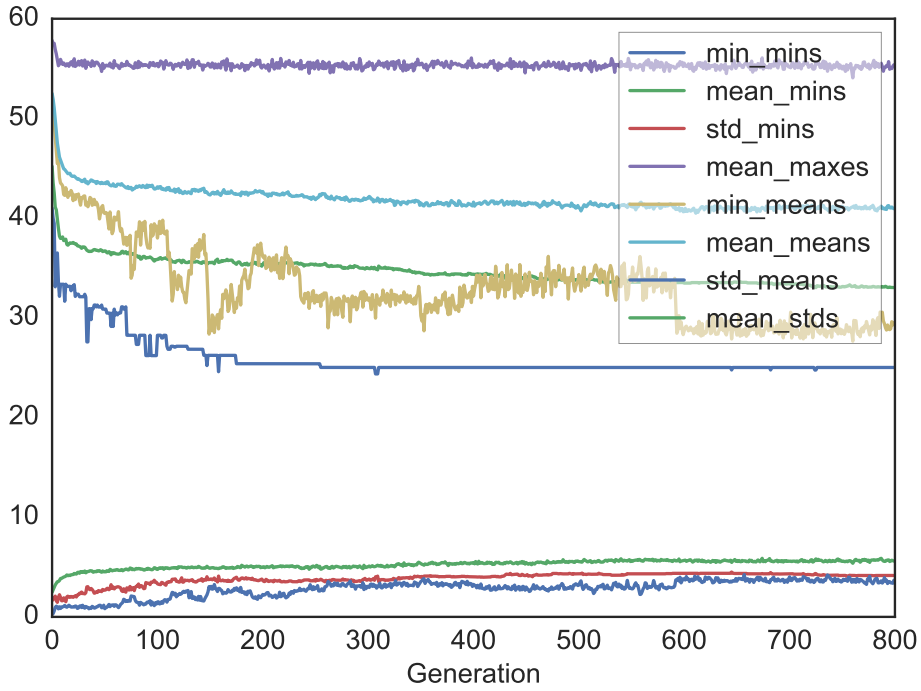
			0	1	2						
			3	4	5						
			6	7	8						
9	10	11	18	19	20	27	28	29	36	37	38
12	13	14	21	22	23	30	31	32	39	40	41
15	16	17	24	25	26	33	34	35	42	43	44
			45	46	47						
			48	49	50						
			51	52	53						

É possível, também, atribuir o fiasco aos operadores genéticos - mutação, principalmente, visto que é o que mais diverge do que foi implementado em outros artigos que obtêm êxito nesta tarefa[4, 1]. A implementação poderia ter sido modificada, por exemplo, para fazer o mesmo que foi feito por Herdy[1] em 1994 e meramente inserir trechos que já tem grande significado semântico. No entanto, isso não foi feito devido à intenção de fazer um algoritmo genético que opera de maneira **justa** - isso é, que não aplique conhecimento de algoritmos determinísticos para a resolução do cubo mágico em seus operadores.

É difícil estabelecer um limiar do que é justo ou não para com algoritmos genéticos em um contexto em que já existe um algoritmo determinístico que resolve o mesmo problema. Devido à grande liberdade que existe quando se modela um GA, existem várias maneiras de incorporar o conhecimento de métodos determinísticos. Uma abordagem possível seria definir como função de fitness o número de passos que o algoritmo determinístico (o *God's Algorithm*[3]) levaria para alcançar a solução. No entanto, embora seja intuitivo que tal fitness funcionaria muito melhor do que a fitness proposta neste trabalho, a partir de que momento o algoritmo genético deixa de ser um algoritmo genético justo e passa a funcionar apenas por mérito do algoritmo determinístico?

O contraposto também é relevante no mesmo sentido, claro - qual o patamar a partir do qual o conhecimento de domínio utilizado para fazer um algoritmo genético considerado justo para o problema do cubo mágico passa a ser uma característica de um algoritmo determinístico? Pelo mesmo critério utilizado para criticar a implementação de Herdy no parágrafo anterior, é possível

Figura 15: Fitness sem elitismo



ser cético quanto ao uso da contagem de *facelets* de cor correta na face dos cubos para a fitness, como feito neste trabalho. Será que essa informação não poderia ser útil para elaborar um algoritmo determinístico?

Por fim, resta apenas considerar que este problema é um mau candidato para a resolução com um algoritmo genético em geral. Não pelos resultados de baixa qualidade obtidos por esta modelagem e interpretação, mas pela existência de formas muito melhores e ineficientes de fazê-lo. Vale mencionar que isso, inclusive, já foi mencionado na resposta melhor avaliada[5] para a pergunta do *StackOverflow* presente na especificação do trabalho, feita pelo usuário "zegkljan". Após alguns parágrafos com formas de implementar um algoritmo genético para resolver o problema do cubo mágico, ele afirma que *"quando se sabe algo sobre o problema (que, no caso do cubo mágico, se sabe), é muito melhor usar um algoritmo dedicado ou pelo menos informado para resolver o problema, ao contrário de um quase cego como um GA."*

9 Conclusão

Neste trabalho, foi planejado e implementado um algoritmo genético com a intenção de resolver instâncias do cubo mágico evolucionariamente. A implementação foi feita em Python de forma modular e extensível, com alguns operadores genéricos e outros específicos para o problema do cubo mágico. Foi modelado e implementado também, consequentemente, o cubo mágico de lado 3 e seus movimentos válidos. Depois disso, foram realizados inúmeros experimentos, variando parâmetros e avaliando suas consequências de acordo com as métricas de qualidade utilizadas.

Ao final, o algoritmo genético utilizado foi incapaz de resolver as instâncias do cubo fornecidas como entrada, mas conseguiu soluções que chegam aparentemente perto de uma solução. Foi discutido, então, o que pode ter causado a ineficácia do método utilizado. Finalmente, foi discutida a viabilidade e praticidade do uso de algoritmos evolucionários para resolver o cubo mágico e encarar outros problemas para os quais já existem métodos determinísticos eficazes.

Figura 16: Fitness da instância *in2*

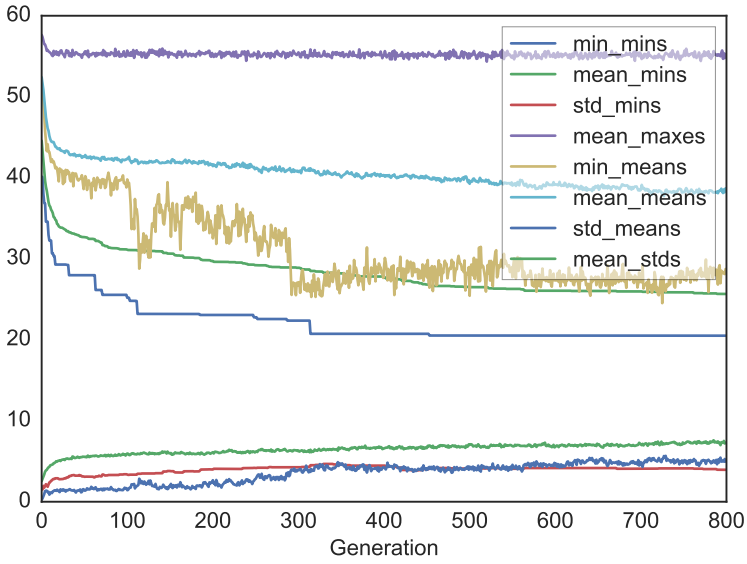
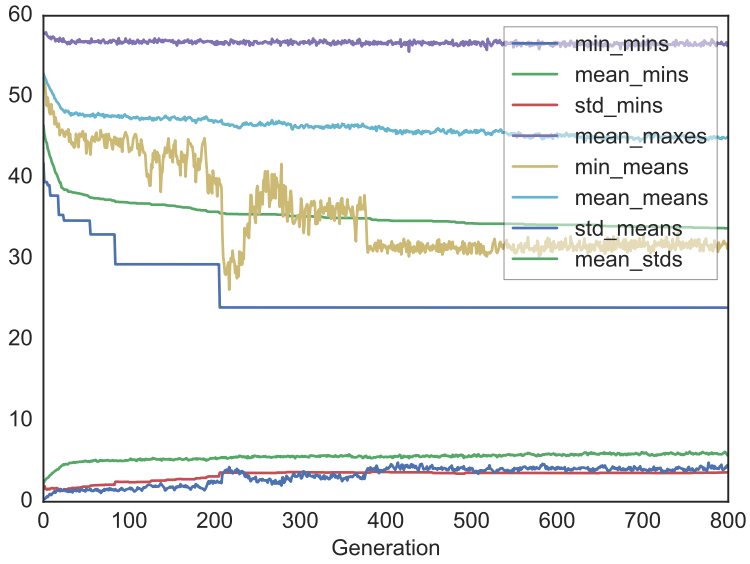


Figura 17: Fitness da instância *in3*



Referências

[1] M. Herdy e G. Patone. "Evolution Strategy in Action, 10 ES-Demonstrations". Em: *Technical Report, International Conference on Evolutionary Computation* (1994).

[2] Eric Jones, Travis Oliphant, Pearu Peterson et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2016-09-12]. 2001–. URL: `http://www.scipy.org/`.

[3] Tomas Rokicki et al. *God's Number is 20*. 2010. URL: `https://sites.google.com/site/semeval2012task2/home` (acedido em 12/09/2016).

[4] Nain El-Sourani e Markus Borschbach. "Design and Comparison of two Evolutionary Approaches for Solving the Rubik's Cube". Em: *Parallel Problem Solving from Nature* (2010). URL: `http://arxiv.org/abs/1310.5042`.

[5] zegkljan. *"Solving a given Rubik's cube by GA", Resposta à pergunta "Rubik's cube genetic algorithm solver?"* 2016. URL: `http://stackoverflow.com/a/36469178` (acedido em 12/09/2016).