Relatório do 1º Trabalho Prático

Felipe Colombelli[1]
Giovanna Lazzari Miotto[1]
Henrique Corrêa Pereira da Silva[1]

[1]Instituto de Informática, Universidade Federal do Rio Grande do Sul
{fcolombelli, glmiotto, hcpsilva}@inf.ufrgs.br

6 de novembro de 2019

Resumo

Neste trabalho, implementamos e validamos três algoritmos de aprendizado de máquina que controlam um carro em uma corrida, tomando decisões a partir de features calculadas com base nos seus sensores ao mundo externo. Nossas implementações não alcançaram os resultados desejados para evitar o *overfitting* do agente para uma pista específica, mas ainda obtivemos resultados interessantes e consideráveis. Acreditamos que os resultados poderiam ser aprimorados se adotado um design de treino levemente modificado para utilizar a mesma pista por um número determinado de vezes antes de trocá-la, além de, é claro, testar para um conjunto diferente de features, como o valor cru dos sensores. No melhor caso, o algoritmo genético concluiu seu treinamento com mais de 15000 pontos para o design de treino adotado. O link para o repositório é o seguinte: `https://bitbucket.org/dieMaus/ai-racers`

Sumário

1 Algoritmos Implementados

Dentre os variados algoritmos que nos foram apresentados tanto em aula quanto nas publicações que versam sobre o estado da arte, escolhemos os sequintes:

- CMA-ES
- Algoritmos Genéticos
- Hill Climbing

Esse grupo de algoritmos, acreditamos nós, é uma gama que tem tanto abordagens clássicas quanto modernas, o que para a comparação futura nos pareceu interessante.

1.1 CMA-ES

O algoritmo/CMA-ES/gera novas populações a partir da amostragem de uma distribuição normal N-dimensional, onde N é o número de pesos, em torno de um ponto N-dimensional correspondente ao candidato médio da geração corrente.

O candidato médio que propagará a atual geração é determinado por média ponderada dos melhores candidatos da geração, ordenados pela fitness demonstrada em episódio de corrida.

1.1.1 Tamanho da População

Entre os poucos parâmetros do CMA-ES, a quantidade de novas soluções amostradas da distribuição normal por geração tem maior impacto.

Os experimentos iniciais com população de 5 amostras foram rápidos, porém não se mostraram capazes de introduzir a diversidade necessária para o progresso do aprendizado, rapidamente sucumbindo ao critério de parada por plateau em pontuação aquém dos demais algoritmos (próximo a 1000). O aumento da população para 20 amostras suscitou um comportamento mais próximo do esperado para uma técnica evolutiva estocástica, embora ainda longe do desejado e com maior custo computacional.

Amostras maiores foram tentadas; o tempo de execução estendeu-se sem ganho proporcional na convergência para pontuações melhores.

1.1.2 Elitismo

A proporção `best_percentage` de candidatos considerados na média é definida via argumento, com *default* em 50%.

Foi observado experimentalmente que um elitismo exacerbado leva a resultados menos satisfatórios. Em função da quantidade elevada de dimensões, um conjunto de pesos com pontuação pior ainda pode agregar boas descobertas à solução média; em um jogo como o *AI Racers*, um peso ruim pode afetar o desempenho do jogador, mesmo que os demais sejam desejáveis.

Assim, um elitismo mais abrangente oferece maior oportunidade de inovação e impede a concentração da próxima geração em um máximo local baixo. Por outro lado, selecionar candidatos demais dilui demais o foco na região que nos interessa.

1.1.3 Implementação

Nossa implementação do CMA-ES foi baseada principalmente na versão apresentada em aula. Embora a versão de aula tenha tornado claro a intuição e funcionamento do algoritmo, algoritmo, o código proposto no paper original [Hansen and Ostermeier, 2001] apresentava características mais robustas - e, por conseguinte, mais complexas - de adaptatividade da matriz de covariância que define a distribuição Gaussiana.

Uma forma de emular essa flexibilidade adaptativa ao longo da execução é alterar o $sigma^2$ multiplicador da matriz de covariância, que, em nossa implementação, ficou constante em 0.5^2.

Por exemplo, a repetição de um mesmo candidato médio ou de uma maioria da piscina de candidatos amostrados por uma ou mais iterações poderia provocar um aumento gradual em sigma, de modo a tornar a Gaussiana mais abrangente e, talvez, introduzir candidatos promissores que tirem o algoritmo do máximo local.

Analogamente, quando o processo precisar tornar a amostragem mais concentrada (por falta de foco da normal), o sigma poderia ser reduzido em torno da média.

1.2 Algoritmos Genéticos

Algoritmos Genéticos correspondem a um grupo de metaheurísticas que aplicam analogias provindas da biologia e da solução natural para selecionar indivíduos mais aptos em determinada situação. Por serem algoritmos clássicos e por possuirem diversas variações e abordagens únicas na literatura, acreditamos que seriam um tópico interessante de se aplicar no trabalho prático.

Os parâmetros considerados na perturbação do algoritmos são: o tamanho da população considerada a cada iteração, a fração de vezes esperada para que se ocorra uma mutação considerado todos os genes de todos os filos, e a fração elitista de indivíduos selecionados para a próxima geração.

1.2.1 Geração da População

A população no algoritmo genético implementado é gerada apenas na primeira iteração através de um sampling uniformemente distribuído sobre um intervalo meio-aberto, $[low, high)$.

Visando aumentar a performance do algoritmo, a geração dos indivíduos é feita em paralelo para o tamanho definido de população.

1.2.2 Recombinação

O *crossover* implementado gera uma máscara aleatória cuja função é selecionar, para cada gene do indivíduo filho, de qual indivíduo pai o herdará.

Os genes aqui considerados são um número real interpretado como um peso. Este peso é associado a cada uma das features implementadas (além do peso de bias) e cada ação possui uma combinação própria de pesos. Um indivíduo, por sua vez, é composto por este conjunto de pesos.

O método de recombinação também foi codificado para que fizesse suas operações em paralelo, aproveitando o poder computacional disponível.

1.2.3 Mutação

Dois tipos de mutação foram implementadas, mas apenas a segunda foi considerada.

O primeiro tipo utiliza a taxa de mutação para sortear, a cada indivíduo filho, um valor indicando se ele sofrerá mutação. Em caso positivo, para cada gene do indivíduo, a mesma taxa é utilizada para sortear se o gene deve ou não sofrer mutação.

Por fim, mais um número real dentro de um range pré-definido como $[-0.5, 0.5]$ é sorteado e será somado ao parâmetro.

O segundo tipo, que foi utilizado nos experimentos, utiliza a taxa de mutação para sortear, a cada gene de todos os indivíduos filhos gerados, um valor indicando se este gene deve sofrer mutação. Em seguida, aplica a perturbação aleatoriamente gerada (para o mesmo range citado acima) ao gene sorteado.

1.2.4 Seleção da Nova Geração

Dois métodos de seleção foram implementados, seleção por roleta e por elitismo. Para a realização dos experimentos, taxa de seleção por roleta foi fixada em 0.1 e a taxa de elitismo variada em 0.2 e 0.5.

A seleção por roleta demandou que o array contendo o *fitness* (score) de cada indivíduo fosse *shifted* (translocado) para zero somando o valor absoluto do menor elemento em todos os elementos. Isso foi feito com o auxílio de um novo *array* que servia apenas para rodar o sorteio, sendo que este *array* mantinha sempre o mesmo número de elementos que o *array* original de *fitness* e que o *array* representando a população. Além disso, o novo *array* existe apenas no escopo do método de seleção para auxiliar no funcionamento da roleta.

Também vale destacar que, para evitar os casos em que o *array* inteiro de *fitness* valesse zero e, portanto, a soma dos seus elementos fosse igual ao valor do menor elemento (limitante inferior = limitante superior = 0), adicionou-se a constante 1 à soma dos elementos.

1.3 Hill Climbing

O algoritmo de hill climbing foi selecionado a fim de providenciar uma baseline e teste de sanidade para os demais.

Há inúmeras variações do algoritmo de hill climbing. Como nosso objetivo era implementar um método básico de busca que contrastasse com os demais algoritmos, o grupo optou por uma geração exaustiva de vizinhos sem qualquer elemento de randomização além da seed inicial.

Aqui, define-se um vizinho como uma cópia da melhor solução atualmente conhecida com modificação em um único peso por uma perturbação +step ou -step determinada por parâmetro e com default 0.5.

A seleção do melhor candidato utiliza o critério de Best Improvement, de forma que a iteração não termina sem avaliar todos os vizinhos.

O vizinho de maior pontuação superior à atual sobrevive como a solução corrente para a próxima geração.

Caso nenhuma perturbação tenha gerado um vizinho mais desejável, o método seria incapaz de prosseguir, tendo atingido máximo local para o step usado.

Para remediar terminações precipitadas, foi adicionado um modificador desespero que multiplica o step por um fator linearmente crescente a cada iteração consecutiva sem melhora na fitness do candidato. O encontro de uma solução melhor retorna o desespero para o valor inicial.

A busca termina ao atingir o máximo de iterações ou o limite de iterações sem melhora, mesmo com desespero.

2 Desenvolvimento

Sendo um trabalho primeiramente de implementação, tentamos colocar em primeiro lugar os interesses e curiosidades do grupo na escolha dos algoritmos.

2.1 Estratégia de Desenvolvimento

Começamos desenvolvendo os algoritmos de treinamento com a utilização de três features para testar e depurar tais algoritmos.

Com o auxílio do *Google Colab*, ferramenta online para utilização de notebooks `.ipynb` equivalentes aos gerados pela ferramenta *Jupyter Notebook*, pudemos realizar testes preliminares na implementação dos algoritmos de treino, assim como investigar melhor como algumas funções (principalmente do `numpy`) funcionavam.

Assim que os códigos ficaram funcionais, treinamos o agente normalmente utilizando loops while e, então, definimos o que seria nosso primeiro conjunto de features.

Percebemos, então, que o agente estava sobreajustando a performance para a pista treinada e não conseguia generalizar para outras pistas. Testamos brevemente o desempenho do agente para outros conjuntos de features e o comportamento persistiu, nos levando a adotar uma nova estratégia de treino que transferia o controle do loop para um script externo.

2.2 Sessões de Treinamento

A abordagem de treino adotada seguiu uma lógica de interação entre um script bash e os códigos implementados em Python.

Todos os algoritmos foram modificados e passaram a salvar o resultado e informações relevantes ao loop de treino, a cada iteração. Tais arquivos eram criados na inicialização da sessão de treino e carregados e atualizados durante esta sessão.

Essa mudança teve que ser propagada ao arquivo `AIRacers.py`, que controlava todos os parâmetros e opções de entrada do programa, a fim de que esse pudesse aceitar também nossos meta-parâmetros como opção de entrada.

Além disso, realizamos mudanças a lógica de herança dos controladores para evitar a repetição do código das features e, assim, adicionamos um *template* extra de controlador, onde realizaríamos as mudanças de *features*.

2.2.1 Estado da Execução

O design de treino, portanto, desacoplou a lógica de iterações do código, que por sua vez rodava apenas um episódio por indivíduo, salvando os resultados intermediários e o estado de execução em arquivos binários próprios.

O trade-off gerado pelo design de treino escolhido confrontou flexibilidade e *overhead* de I/O. Em troca de uma sessão de treino flexível que pudesse permutar combinações de pista e agente inimigo a cada iteração, gerou-se um custo computacional adicional para salvar e carregar os dados no disco rígido.

2.2.2 Script de Treinamento

O script bash gerado para realizar as iterações de treino selecionava uma combinação de pista e agente inimigo disponíveis para treino e executava o código em modo *learn* para esta combinação.

Conforme execução do programa, o script realizava o *parse* da saída gravada, coletando ambas pontuação e pesos ótimos encontrados na iteração de aprendizado.

2.2.3 Drawbacks e Balanço das Decisões

Acreditamos, porém, com embasamento empírico obtido durante as sessões de treino acopladas e desacopladas do código Python, que o gargalo não estava no acessos ao disco e sim no método que executava um episódio para avaliar o score obtido.

O algoritmo mais demorado foi o algoritmo genético e, por isso, foram feitas algumas melhorias neste, paralelizando a execução de certos métodos utilizados pelo algoritmo. No entanto, o método responsável por calcular o score dos indivíduos não pôde ser paralelizado por razões de funcionamento do simulador.

2.3 Features Escolhidas

Devido à importância de ter um bom conjunto de features para qualquer algoritmo de busca ser efetivo, e buscando aquele set que melhor descrevesse as informações relevantes sobre a corrida para a controladora autodidata, o grupo tentou, ao longo do desenvolvimento, três conjuntos de features com as quais realizar os experimentos de perturbação dos parâmetros.

- **Conjunto 1**
  1. `diffCheckpoint`
  2. `riskFrontalCollision`
  3. `riskLeftCollision`
  4. `riskRightCollision`
  5. `centralizedPosition`
- **Conjunto 2**
  1. `diffCheckpoint`
  2. `riskFrontalCollision`
  3. `riskLeftCollision`
  4. `riskRightCollision`
- **Conjunto 3**
  1. `diffCheckpoint`
  2. `uncentered`
  3. `needForBrakes`
  4. `enemyThreat`

```
diffCheckpoint = current.distCheckpoint - previous.distCheckpoint
```

Sendo a única constante em todos os conjuntos, o grupo julgou clara a importância da feature `diffCheckpoint` como principal indicador de progresso do jogador na corrida. Cada *checkpoint* passado confere pontos ao jogador, logo um robô ciente disso ganhará mais pontos.

Um progresso em relação ao próximo objetivo gera um *diff* negativo, enquanto um retrocesso na direção contrária torna a *feature* positiva.

Mais ainda, o grupo observou que cada *frame* corresponde a um movimento de 10% da velocidade do veículo, de modo que a maior diferença em `diffCheckpoint` entre frames equivaleria a 10% da velocidade máxima (200.0).

Logo, enquanto as medições do sensor para distância ao *Checkpoint* não têm limite superior, a sua diferença é limitada pelo *horsepower* do veículo e varia entre -20.0 e $+20.0$, dependendo da direção e velocidade. Isso permite uma normalização adequada entre -1.0 e $+1.0$:

```
norm(diffCheckpoint) = 2 * (diffCheckpoint + 20) / 40 - 1
```

No entanto, essa lógica contém uma exceção: ao cruzar um *checkpoint*, a distância do carro ao próximo *checkpoint* passará de um valor próximo de zero a um valor potencialmente centenas de unidades maior, quebrando a normalização.

Para remediar isso, foi incluída uma condicional que limita a medida da *feature* a $+20$ ou -20 antes da normalização.

```
riskFrontalCollision = (1 - onTrack) * 200 +
↪  onTrack * speed / distCenter
```

A *feature* `riskFrontalCollision` foi formulada para ser um indicador de risco relativo de acelerar e de necessidade de desacelerar.

A ideia é que, caso o carro esteja na pista, uma velocidade elevada dividida por uma distância frontal até alguma região gramínea seja sinal de alerta para a controladora, que deverá desacelerar de imediato. Por outro lado, tendo pleno espaço ou baixa velocidade, o veículo pode prosseguir a velocidades superiores sem perigo.

Se o agente já estiver na grama, o risco é levado ao máximo através do sinal binário `onTrack`.

Embora gerenciar a velocidade do agente fosse a intenção por trás dessa *feature*, a performance resultante do treinamento mostrou-se excessivamente respeitosa dos limites urbanos de velocidade.

Como a velocidade varia de 10 a 200, a `distCenter` de 1 a 100, e `onTrack` é binário, a `riskFrontalCollision` pré-normalização pode atingir um máximo de 200 e um mínimo de $1/10$.

```
norm(riskFrontalCollision) = 2 *
↪  (riskFrontalCollision - 0.1) / 199.9 - 1
```

As *features* `riskLeftCollision` e `riskRightCollision` são análogas, e buscando informar a controladora da necessidade de curvas à esquerda ou à direita por proximidade às margens laterais da pista.

```
riskLeftCollision = (1 - onTrack) * 200 + onTrack
↪  * speed / distLeft
norm(riskLeftCollision) = 2 * (riskLeftCollision -
↪  0.10) / 199.9 - 1

riskRightCollision = (1 - onTrack) * 200 + onTrack
↪  * speed / distRight
norm(riskRightCollision) = 2 * (riskRightCollision
↪  - 0.10) / 199.9 - 1
```

Em seguida, foi definida a *feature* `centralizedPosition`, que busca especificamente manter o veículo centralizado na pista, utilizando a diferença absoluta entre a distância para margem esquerda e à margem direita. Como a maior diferença absoluta é de 99 ($100 - 1$), e a menor 0, a feature normalizada fica:

```
norm(centralizedPosition) = (-1 * abs(distLeft -
↪  distRight) + 99) / 99
```

As *features* acima configuram o primeiro conjunto de informações avaliado. Embora esse conjunto tenha tido bom desempenho na competição da turma - treinado usando o algoritmo genético - e, retroativamente, o grupo tenha constatado seu relativo sucesso para o treinamento comparado a outras variações das *features*, algumas falhas comportamentais nos levaram a buscar alternativas.

Em particular, nesse conjunto o agente parecia relutante em aumentar velocidade em situações seguras e tinha uma preocupação excessiva em se manter equilibrado no centro da pista, oscilando com movimentos senoidais em torno do eixo central.

Buscando mitigar isso, foi proposta a exclusão da *feature* `centralizedPosition`, que, por utilizar valores absolutos, parecia redundante junto com os riscos de colisão esquerda/direita. No entanto, os resultados não foram melhores.

O último conjunto de *features* foi elaborado em resposta às deficiências constatadas com o primeiro superconjunto. Particularmente, o grupo buscou consolidar métricas aparentemente redundantes de permanência na pista a fim de acelerar o procedimento de aprendizado e permitir a inclusão de *features* novas relativas ao *bot* adversário.

A `diffCheckpoint` foi considerada essencial e insubstituível. Já a tripla `riskLeftCollision`, `riskRightCollision` e `centralizedPosition` foi julgada reduzível à *feature* simplificada `uncentered`.

```
uncentered = (distLeft - distRight) / 100
```

Essa nova *feature*, já normalizada entre -1 e $+1$ considerando a possibilidade de distância zero em grama, indica de uma só vez a necessidade de dobrar à esquerda ou à direita devido ao sinal. Assim, idealmente, os pesos aprendidos para as ações "L" e "R" poderiam ser espelhados.

No entanto, é importante notar que essa métrica tem problemas se ambas as distâncias forem zero. Uma correção, não implementada para os experimentos abaixo, testaria a ocorrência de distância zero ou de `onTrack` falso, atribuindo os extremos $+1$ e -1 para os casos de exceção, e $(-1, +1)$ para os demais na pista.

Para medir novamente a situação da velocidade, a antiga `riskFrontalCollision` foi simplificada e normalizada para $[-1, 1]$ de modo que servisse às ações de acelerar e frear simetricamente.

```
needForBrakes = speed / (1 + distCenter)
norm(needForBrakes) = 2 * (needForBrakes - 10 /
↪  101) / (200 + 10 / 101) - 1
```

A última novidade do terceiro conjunto de *features*: uma métrica para a ameaça relativa do *bot* adversário na pista. Enquanto todas as *features* anteriores tentavam praticar a condução segura, mantendo-se na pista e regulando a velocidade, esta medida busca tornar o carro sensível à posição do inimigo ao seu redor. O objetivo dela é sinalizar à controladora da necessidade de acelerar para efetuar uma ultrapassagem.

Para isso, a *feature* usa o seno do ângulo do adversário em relação ao jogador e a distância para representar o risco relativo oferecido pela presença do *bot* - note que caso o inimigo esteja atrás, o risco

é inversamente proporcional à distância, ao contrário do caso frontal. Caso o *bot* não esteja próximo (ou seja, a distância do inimigo é superior a 100), a ameaça recebe valor -1, o mínimo valor após a normalização.

```
if enemyDetected and abs(enemyAngle) > 90:
  enemyThreat = abs(sine(enemyAngle)) * (101 -
  ↪  distEnemy)
if enemyDetected and abs(enemyAngle) <= 90:
  enemyThreat = abs(sine(enemyAngle)) * (1 +
  ↪  distEnemy )

if enemyDetected:
  norm(enemyThreat) := 2 * (enemyThreat - 0) / 101
  ↪  - 1
if not enemyDetected:
  norm(enemyThreat) := - 1
```

3 Resultados

Conforme as estratégias citadas em 2.2, realizamos a análise da pontuação obtida a partir de diversas configurações de meta-parâmetros dos algoritmos.

Os resultados que obtivemos nos deram tanto *insight* para realizar mudanças sobre o funcionamento dos algoritmos implementados quanto nos permitiram observar e chegar a uma conclusão sobre as melhores opções dado o ambiente de aplicação.

3.1 Visualizações

Realizamos o mesmo conjunto de visualizações para cada conjunto de *features*, a fim de observar melhor o impacto de cada conjunto.

As imagens se encontram nas próximas páginas, já que sua visualização é facilitada se elas ocuparem todo o espaço da página.

3.2 Primeiro *Set* de *Features*

Para os experimentos realizados com *Hill Climbing* e *CMA-ES*, podemos observar a presença de um sanfonado na sua curva de aprendizado. Isso se deve, principalmente, ao design de treino adotado que, a cada iteração, modifica a pista sobre a qual o agente deve correr. Percebe-se, no entanto, que os valores limítrofes deste sanfonado tendem a aumentar, correspondendo a uma melhora geral do agente para as pistas consideradas.

Para o algoritmo genético, que calcula o score da população em todas as iterações (mesmo para os indivíduos que já haviam sido avaliados), observamos um efeito cordilheira na curva de aprendizado, onde, para a configuração de 20 indivíduos, foi percebida a introdução de melhorias, dado o crescente aumento da cordilheira.

3.3 Segundo *Set* de *Features*

Uma das nossas hipóteses para a ocorrência de *overfitting* do modelo era que a *feature* de centralização pesava muito na tomada de decisão do agente, culminando em ações como andar em loop no primeiro canteiro da `many_forks`. De fato, o *plot* dos gráficos para o segundo conjunto de *features* considerado (que é exatamente igual ao primeiro, porém sem a presença da *feature* de centralização) foi bastante relevante para discutir esta hipótese.

Podemos notar que as curvas de aprendizado para o segundo conjunto de *features* considerado, quase eliminaram o efeito sanfona para o caso do *Hill Climbing*. Em genéticos, houve inclusive um *breakthrough* para uma pontuação altíssima, considerando 0.5 como taxa de mutação. Acreditamos que esta taxa de mutação elevada é essencial para um bom funcionamento do algoritmo genético, pois tira o peso de encontrar um boa solução utilizando apenas as combinações de números aleatoriamente gerados.

Para o *CMA-ES*, podemos notar o quanto uma grande variedade de indivíduos é necessária para que se haja uma melhora significativa. Com 5 indivíduos, o algoritmo oscilou bastante sua performance, mas com 20 conseguiu uma estabilidade melhor.

3.4 Terceiro *Set* de *Features*

O terceiro conjunto de *features*, por fim, conseguiu os melhores resultados. Todos os algoritmos foram capazes de rapidamente encontrar uma boa solução. Mais uma vez, o efeito sanfona quase desapareceu dos gráficos do *Hill Climbing* e podemos notar que a falta de uma numerosa população, tanto para genéticos, quanto para o *CMA-ES*, foi decisiva na oscilação e inabilidade de aprender.

Devido ao retorno da *feature* de centralização e ainda assim diminuição do sanfonado, podemos repensar se de fato esta *feature* influenciava significativamente no *overfitting*. A resposta, no entanto, pode ser a nova normalização que agora considera um limitante inferior de -1 ao invés de zero, espelhando as ações de dobrar à direita e dobrar à esquerda.

3.5 Treinamento em Interlagos

Consideramos importante, também, realizar o treinamento do agente utilizando um design de treino padrão que treinasse em apenas uma pista. As Figuras 16, 17 e 18 mostram os desempenhos dos 3 algoritmos implementados para um treino de 64 iterações na pista interlagos utilizando o oponente `ninja_bot` e considerando os três conjuntos de features definidos.

4 Conclusão

No presente trabalho, tivemos a oportunidade de avaliar como as diversas perturbações de parâmetros para cada algoritmo influencia seu desempenho, reunindo informações para validar algumas hipóteses. Por exemplo, quando temos os parâmetros de população com valores baixos para algoritmos genéticos e *CMA-ES*, a performance decai e oscila muito (na mudança de pistas).

Além disso, também pudemos avaliar como diversos conjuntos de *features* afetam o aprendizado tanto na velocidade em que convergem, quanto no *overfitting* a que o agente é submetido, questionando ainda os reais efeitos de uma *feature* que centraliza o carro na pista. Acreditamos que mais testes seriam necessários para de fato afirmar se tal *feature* tende ou não a sobre-ajustar o modelo.

E, por último, conseguimos avaliar ainda os efeitos de um design de treinamento personalizado para alterar, a cada iteração, as pistas e agentes inimigos disponíveis. Comportamentos bastante interessantes, como os efeitos de sanfona e cordilheira, foram plotados representando as mudanças geográficas proporcionadas por cada pista.

Acreditamos que o aprendizado teria se beneficiado se considerasse o treino na mesma pista por um determinado número de vezes, antes de realizar a troca de pista. Questionamos, também, a qualidade das *features*, faltando uma análise mais detalhada para um conjunto de *features* mais simples que considerassem os dados puros dos sensores do carro.

Como consta no resumo, segue o link para o repositório: `https://bitbucket.org/dieMaus/ai-racers`

Referências

[Hansen and Ostermeier, 2001] Hansen, N. and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.*, 9(2):159–195.

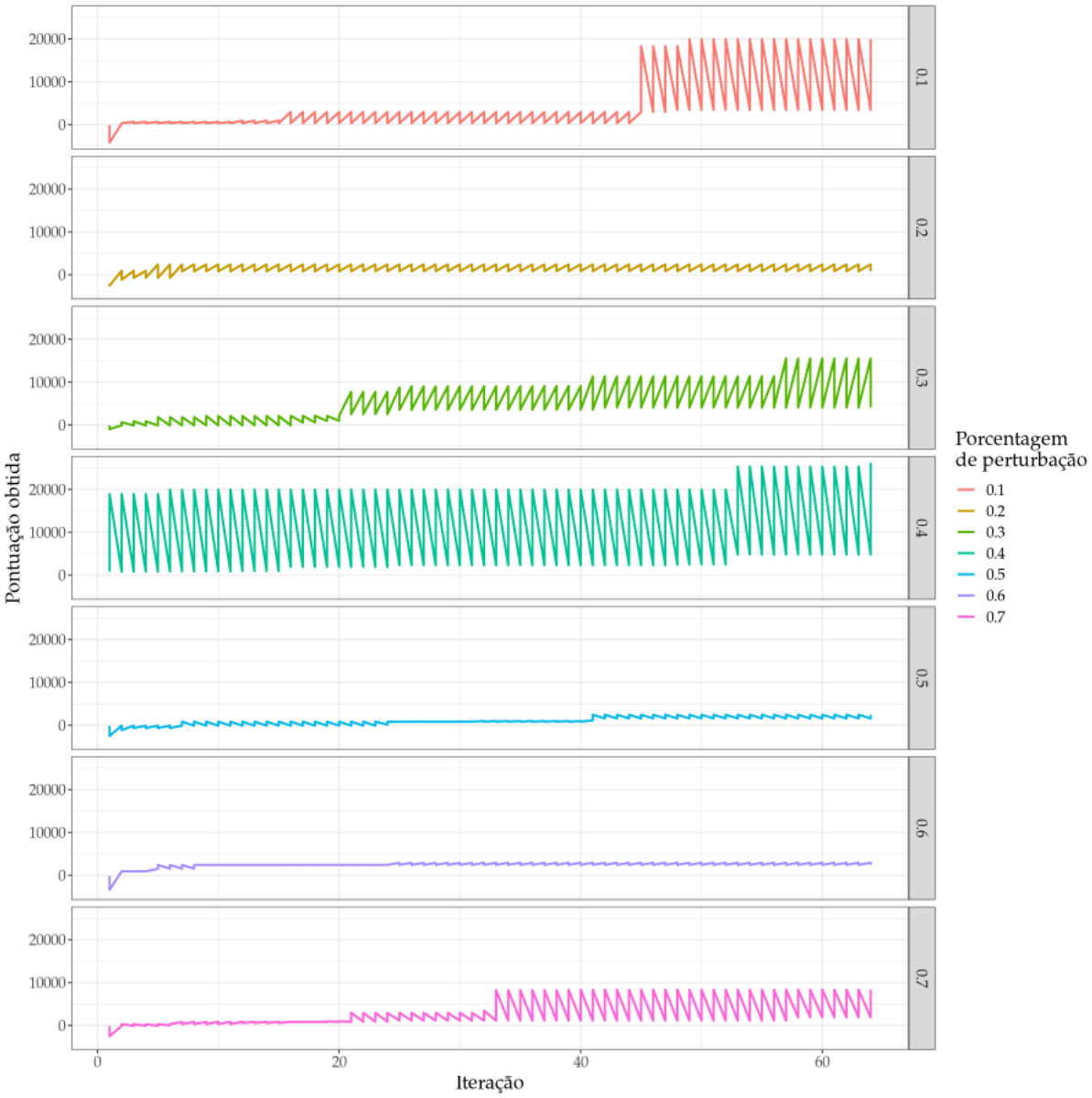

Figura 1: Comparativo de pontuação do HC para o 1º conjunto de *features*

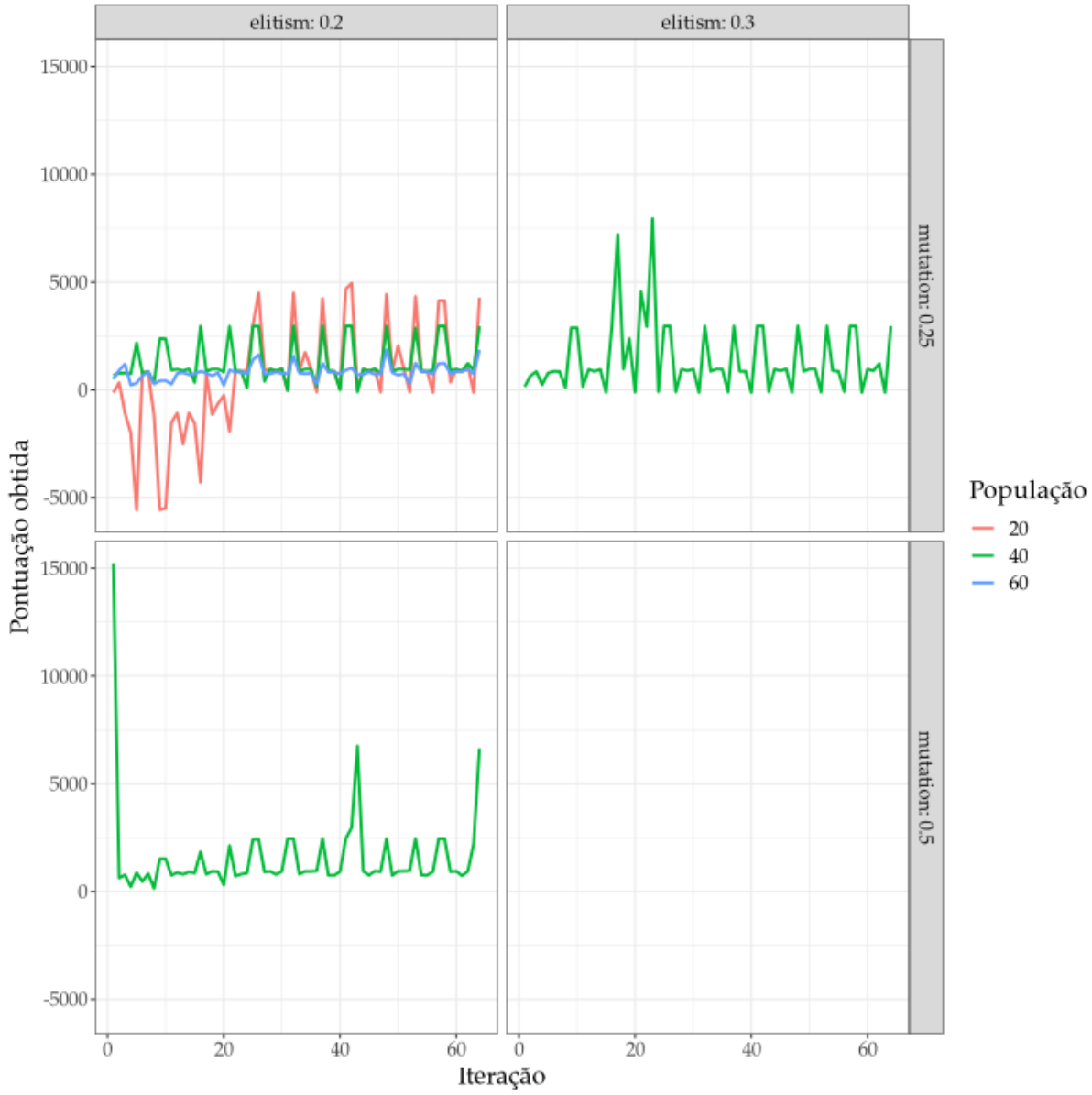

Figura 2: Comparativo de pontuação do GA para o 1º conjunto de *features* conforme nível de elitismo e de mutação utilizados

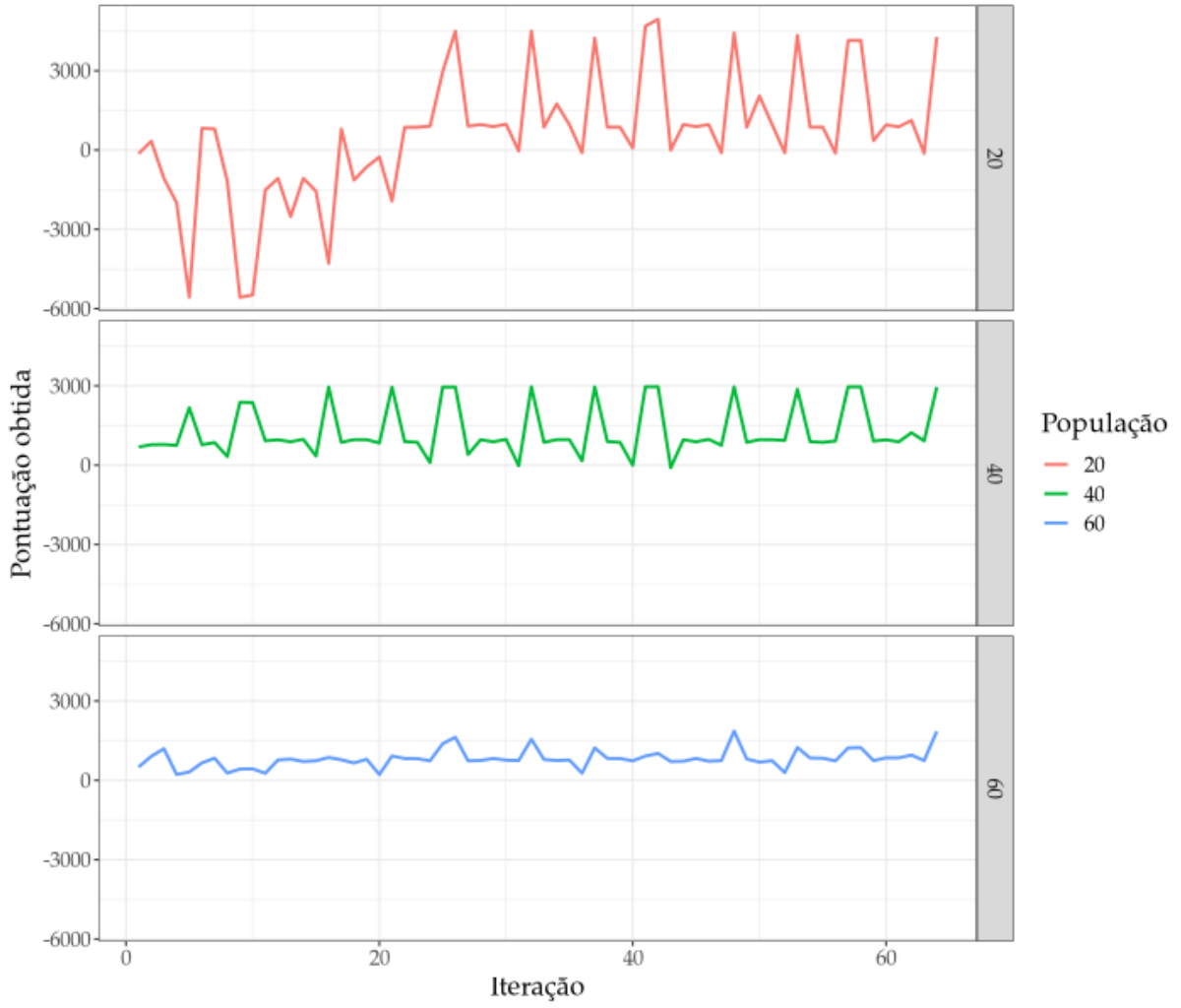

Figura 3: Visualização da pontuação conforme população utilizada para o 1º conjunto de features, com eletismo e mutação fixados a 0.2 e 0.25, respectivamente

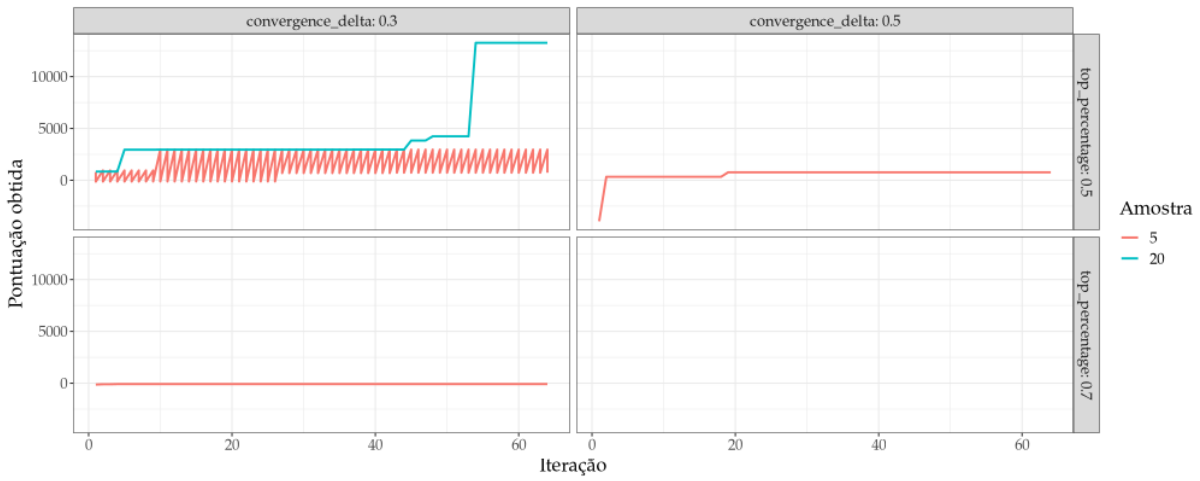

Figura 4: Comparativo de pontuação do/CMA-ES/para o 1º conjunto de features

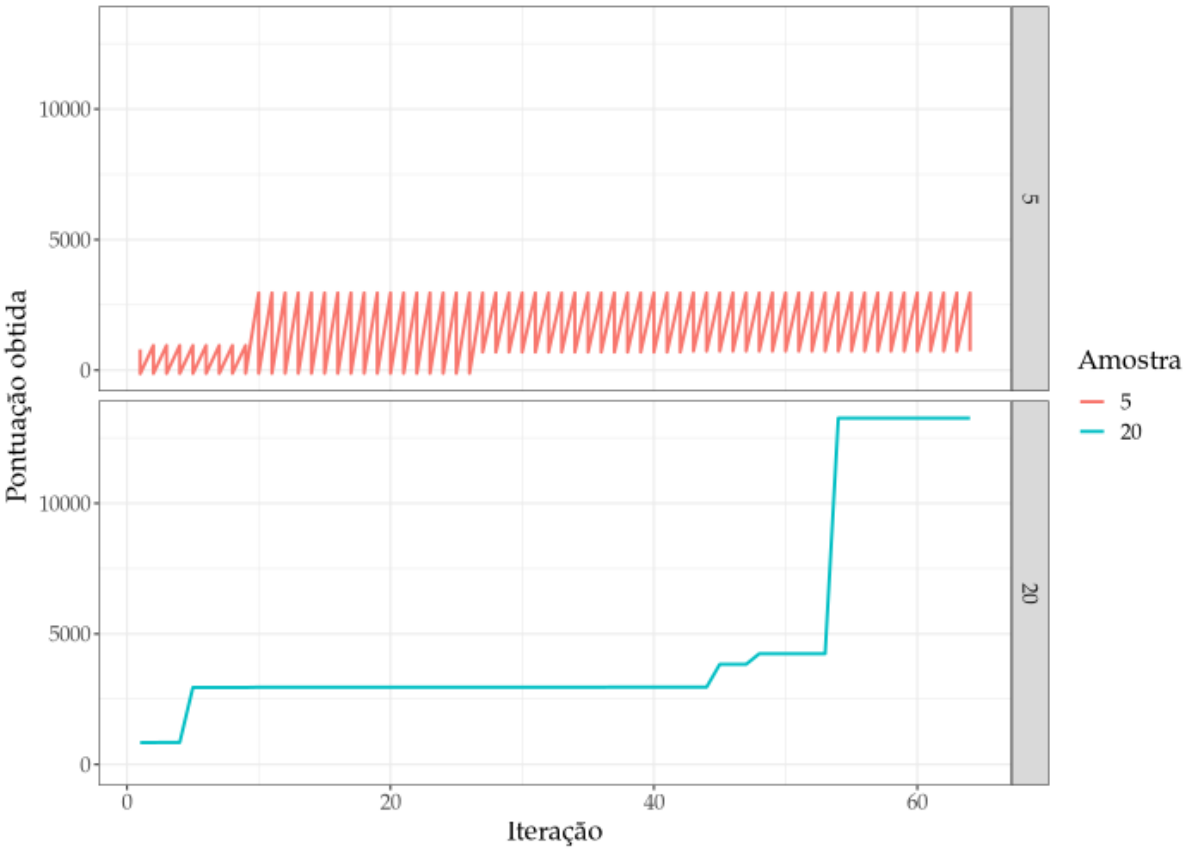

Figura 5: Visualização da pontuação do/CMA-ES/conforme tamanho de amostra para o 1º conjunto de features, com delta de convergência e elitismo fixados em 0.3 e 0.5, respectivamente

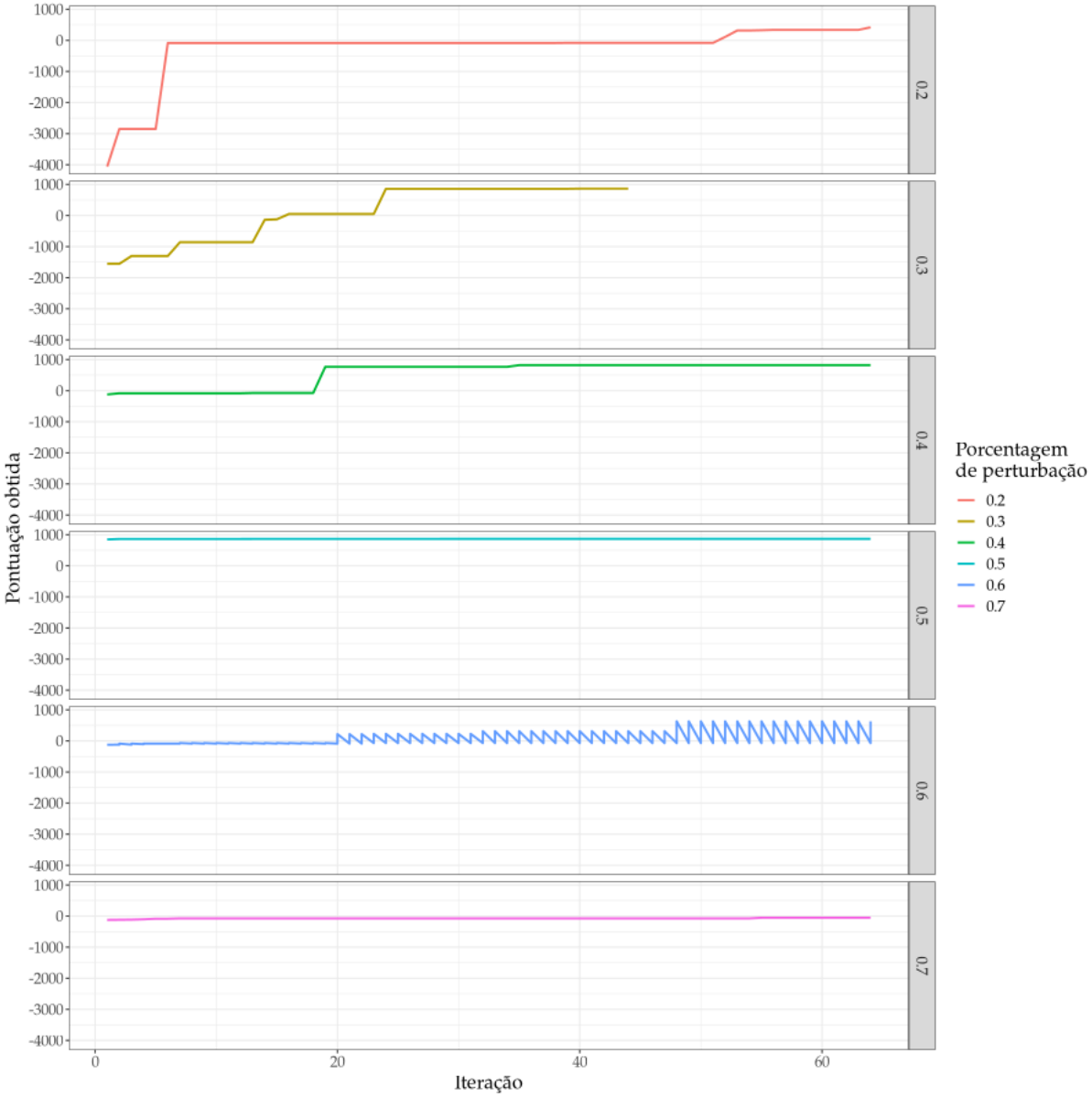

Figura 6: Comparativo de pontuação do HC para o 2º conjunto de *features*

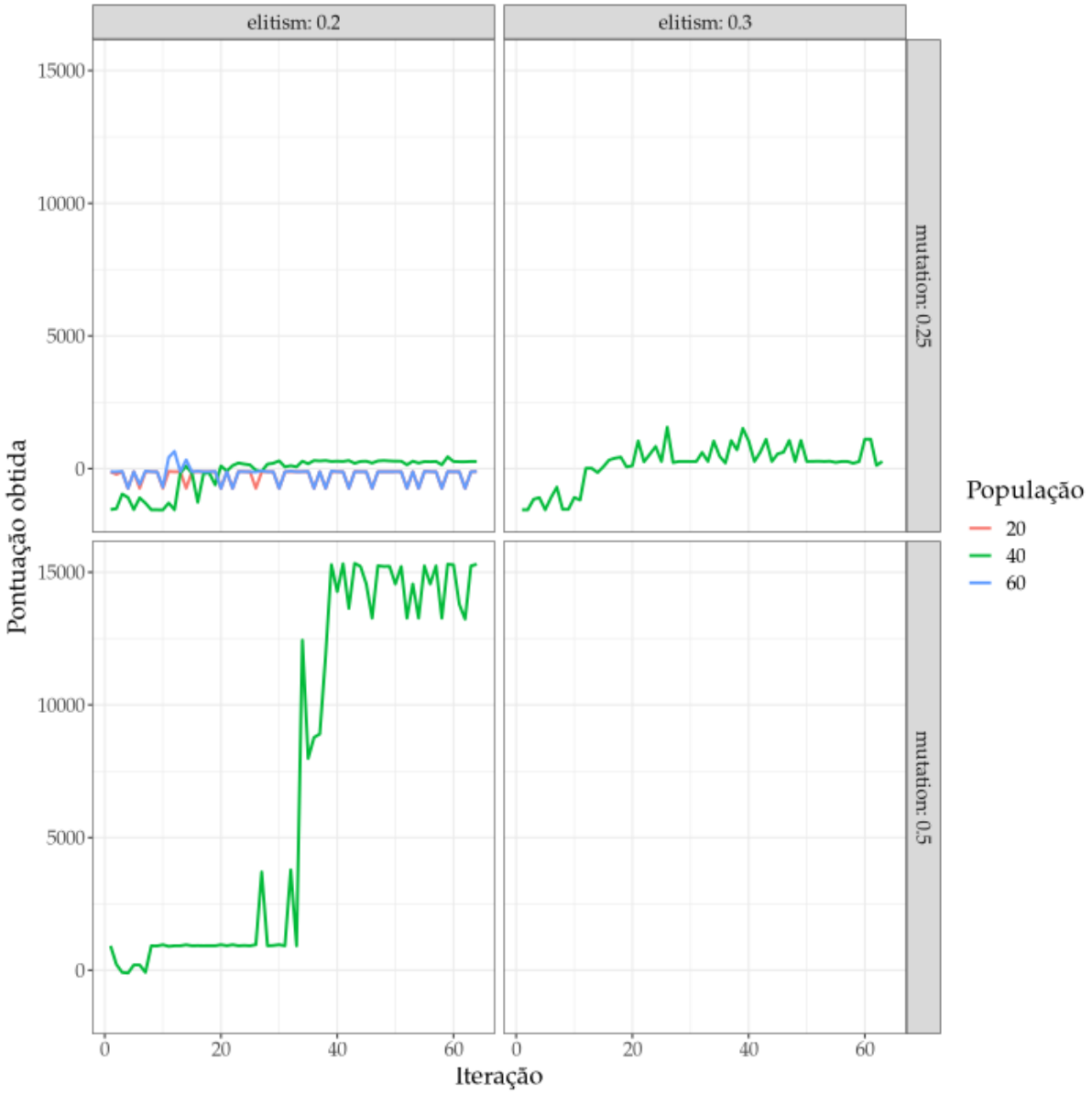

Figura 7: Comparativo de pontuação do GA para o 2º conjunto de *features* conforme nível de elitismo e de mutação utilizados

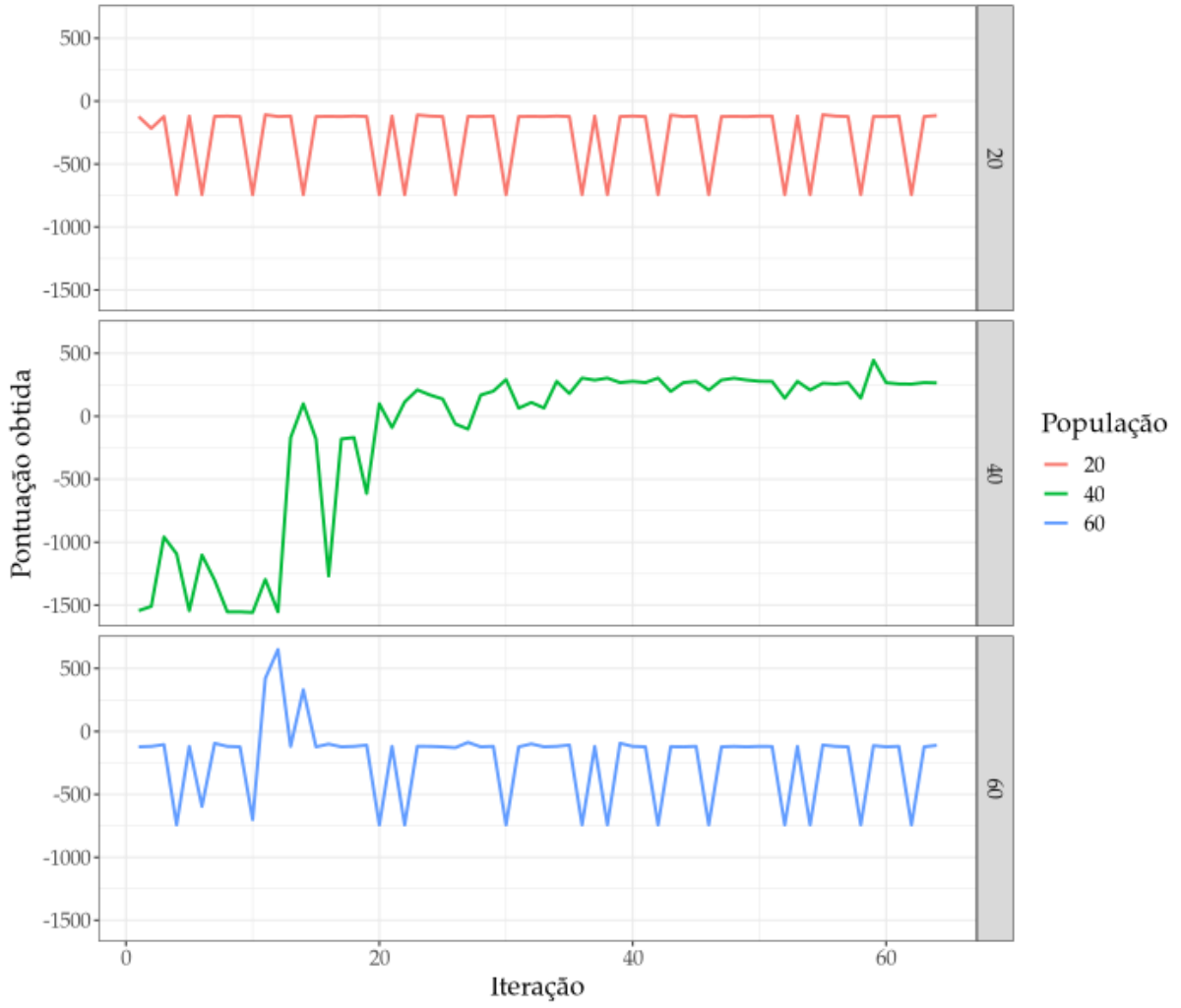

Figura 8: Visualização da pontuação conforme população utilizada para o 2º conjunto de features, com eletismo e mutação fixados a 0.2 e 0.25, respectivamente

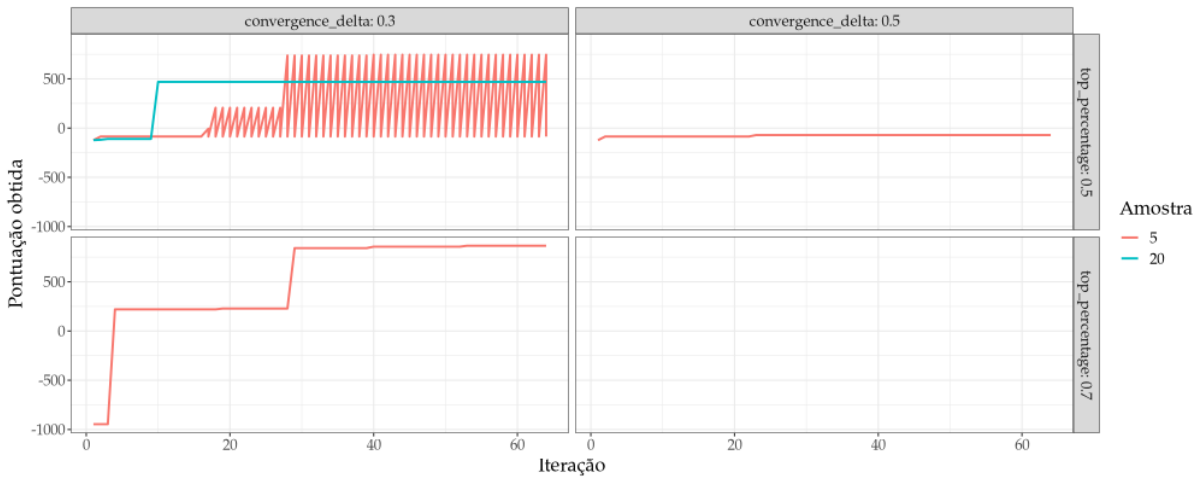

Figura 9: Comparativo de pontuação do/CMA-ES/para o 2º conjunto de features

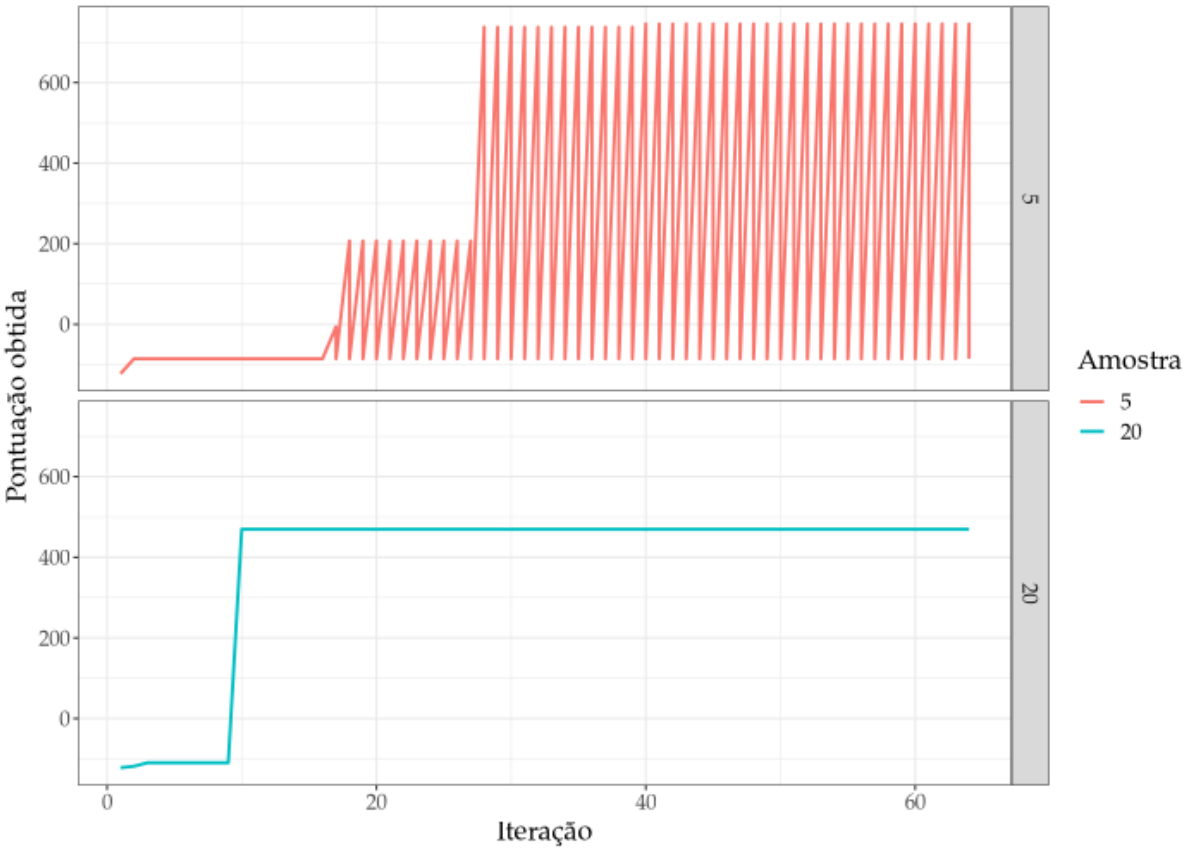

Figura 10: Visualização da pontuação do/CMA-ES/conforme tamanho de amostra para o 2º conjunto de features, com delta de convergência e elitismo fixados em 0.3 e 0.5, respectivamente

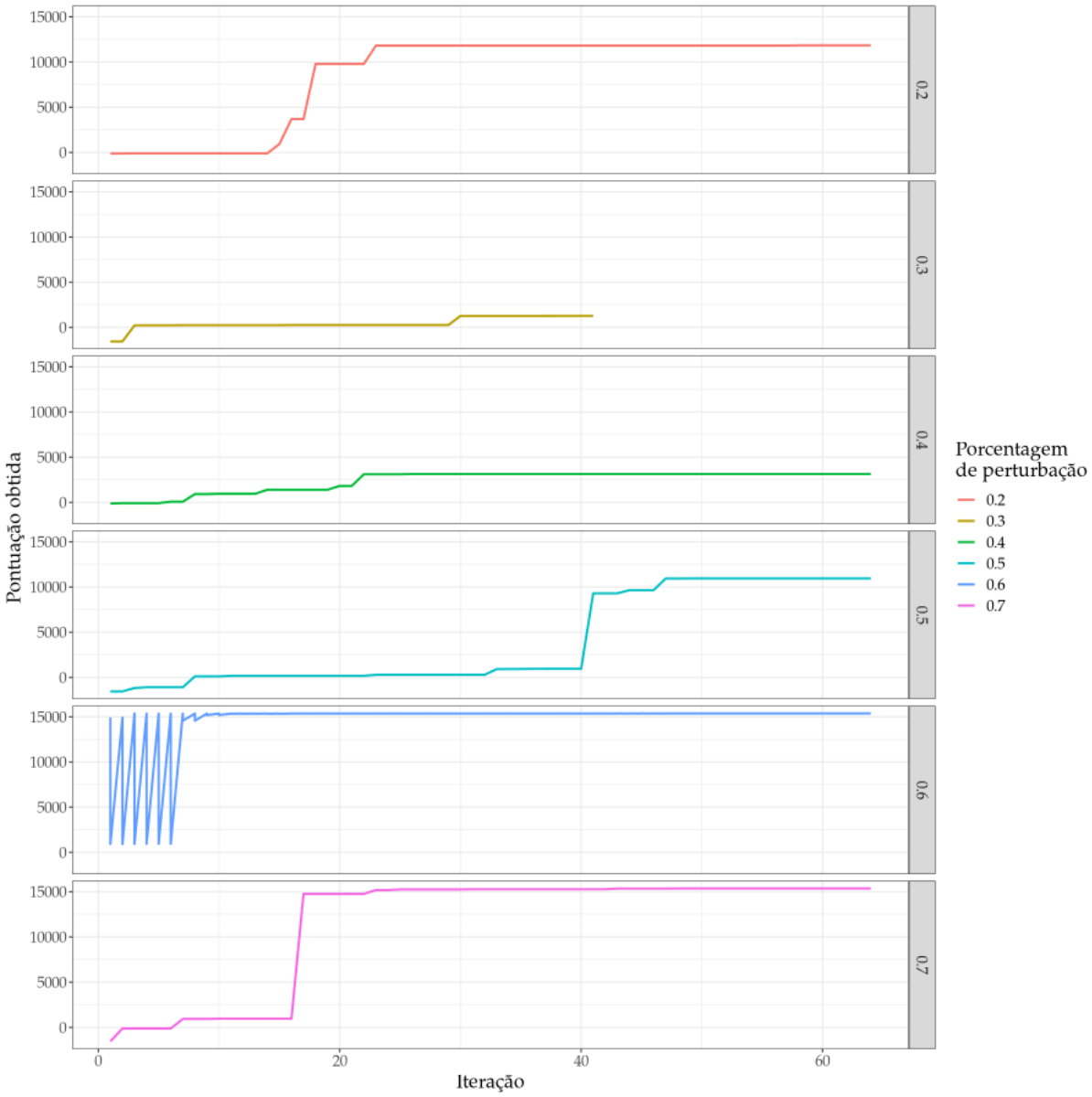

Figura 11: Comparativo de pontuação do HC para o 3º conjunto de *features*

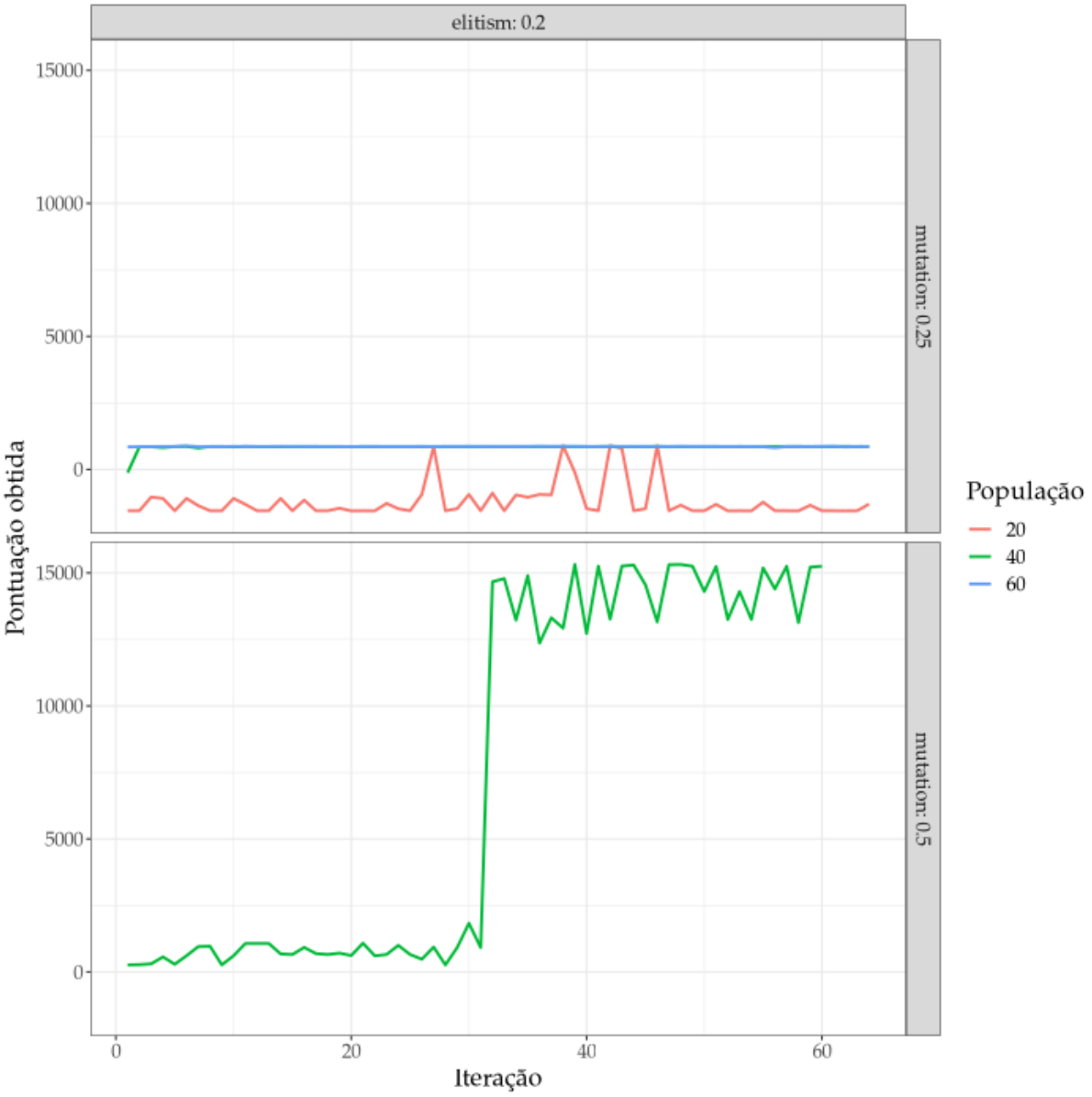

Figura 12: Comparativo de pontuação do GA para o 3º conjunto de *features* conforme nível de elitismo e de mutação utilizados

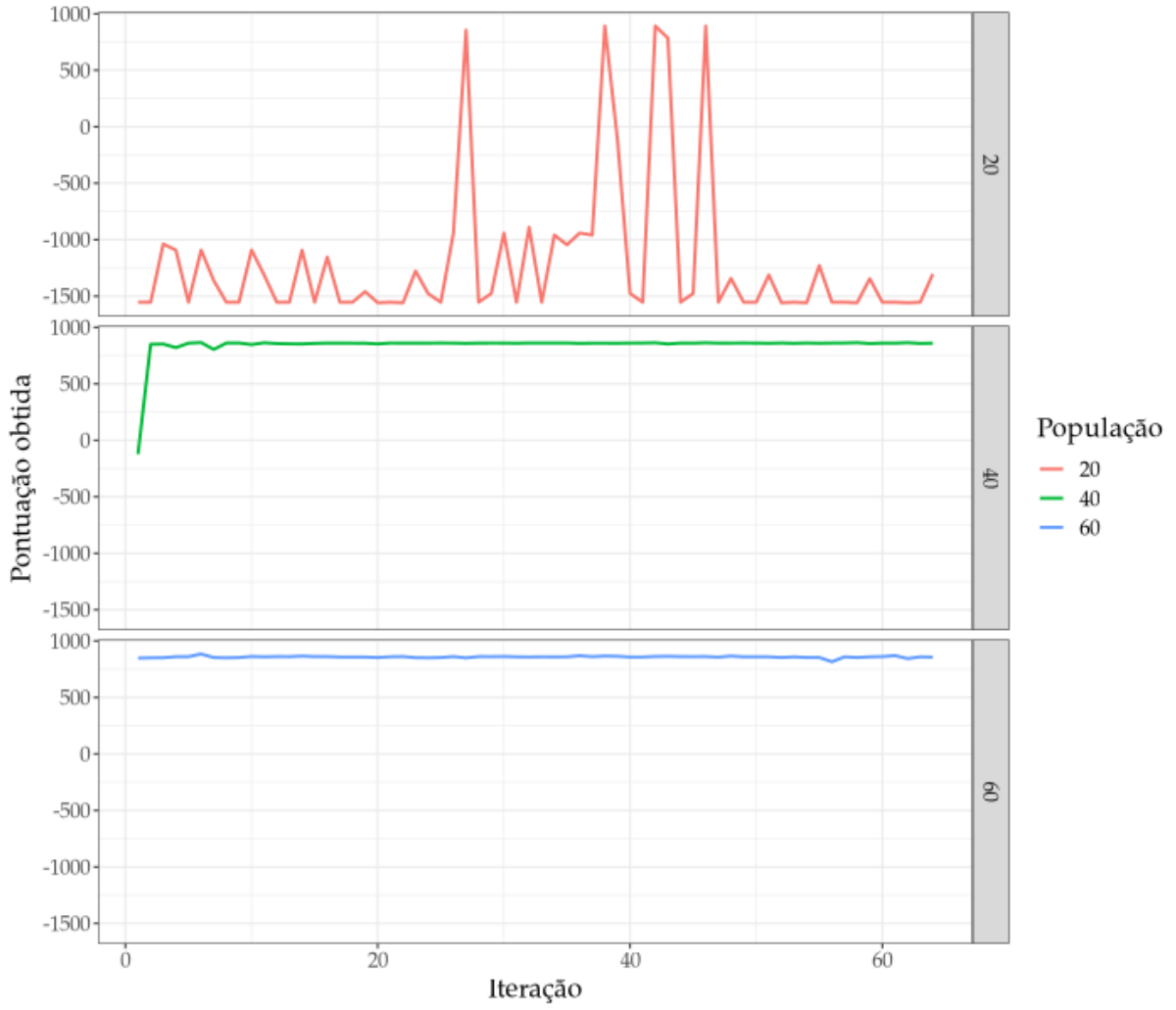

Figura 13: Visualização da pontuação conforme população utilizada para o 3º conjunto de features, com eletismo e mutação fixados a 0.2 e 0.25, respectivamente

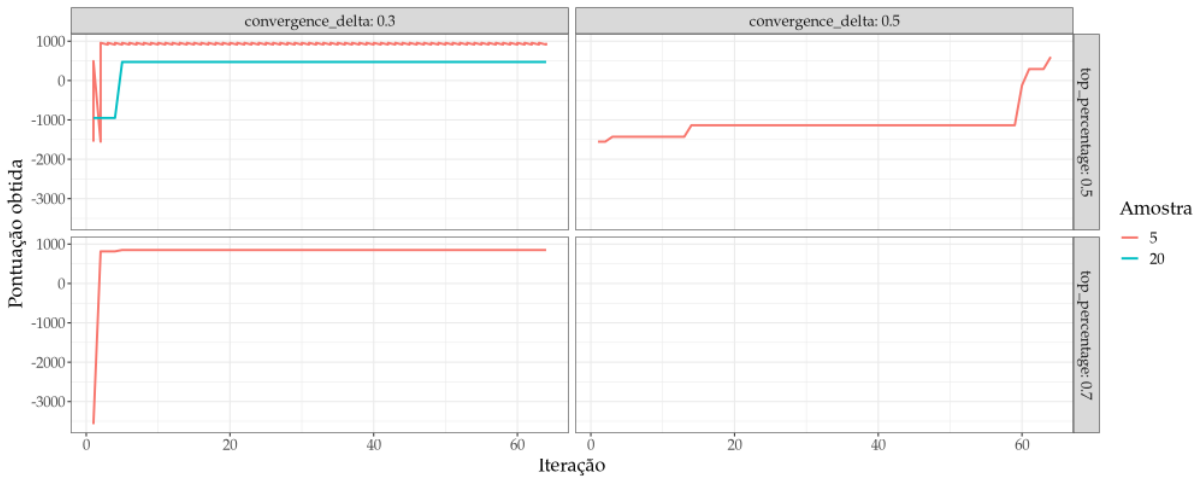

Figura 14: Comparativo de pontuação do/CMA-ES/para o 3º conjunto de features

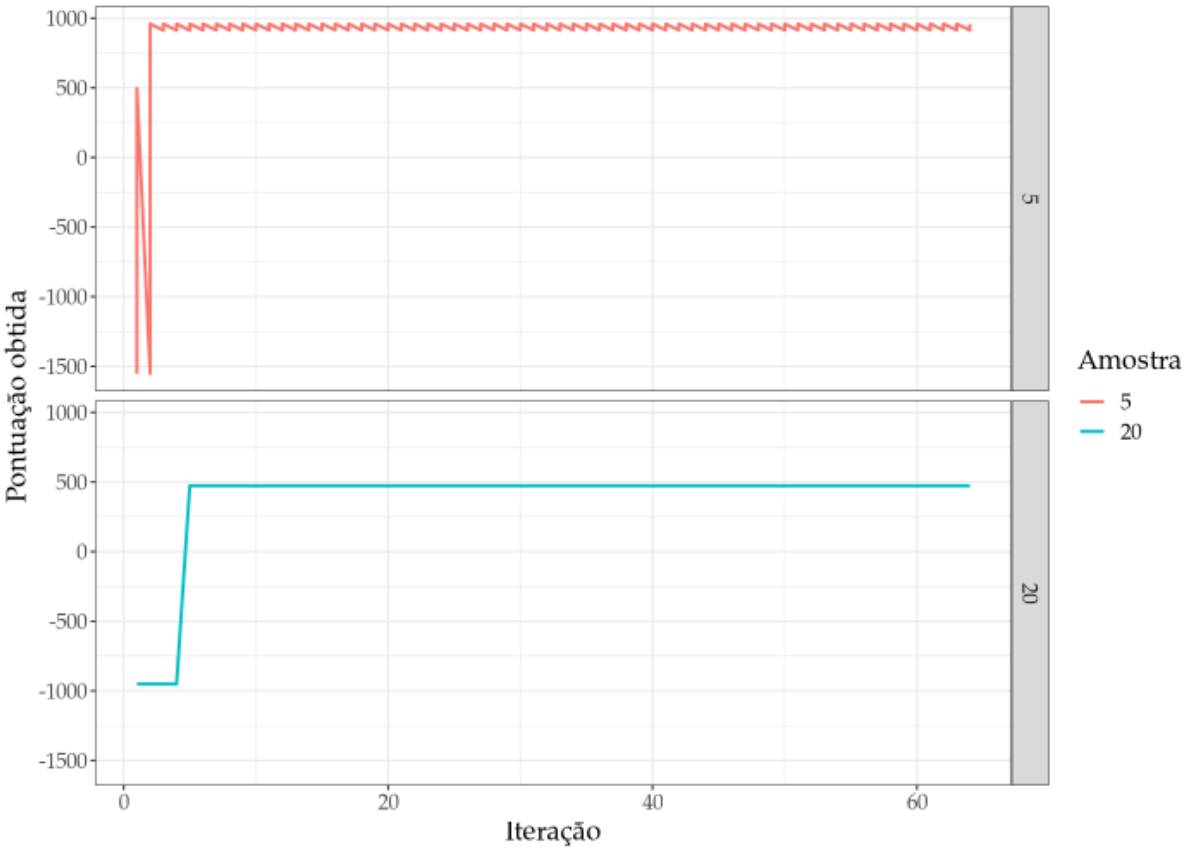

Figura 15: Visualização da pontuação do/CMA-ES/conforme tamanho de amostra para o 3º conjunto de features, com delta de convergência e elitismo fixados em 0.3 e 0.5, respectivamente

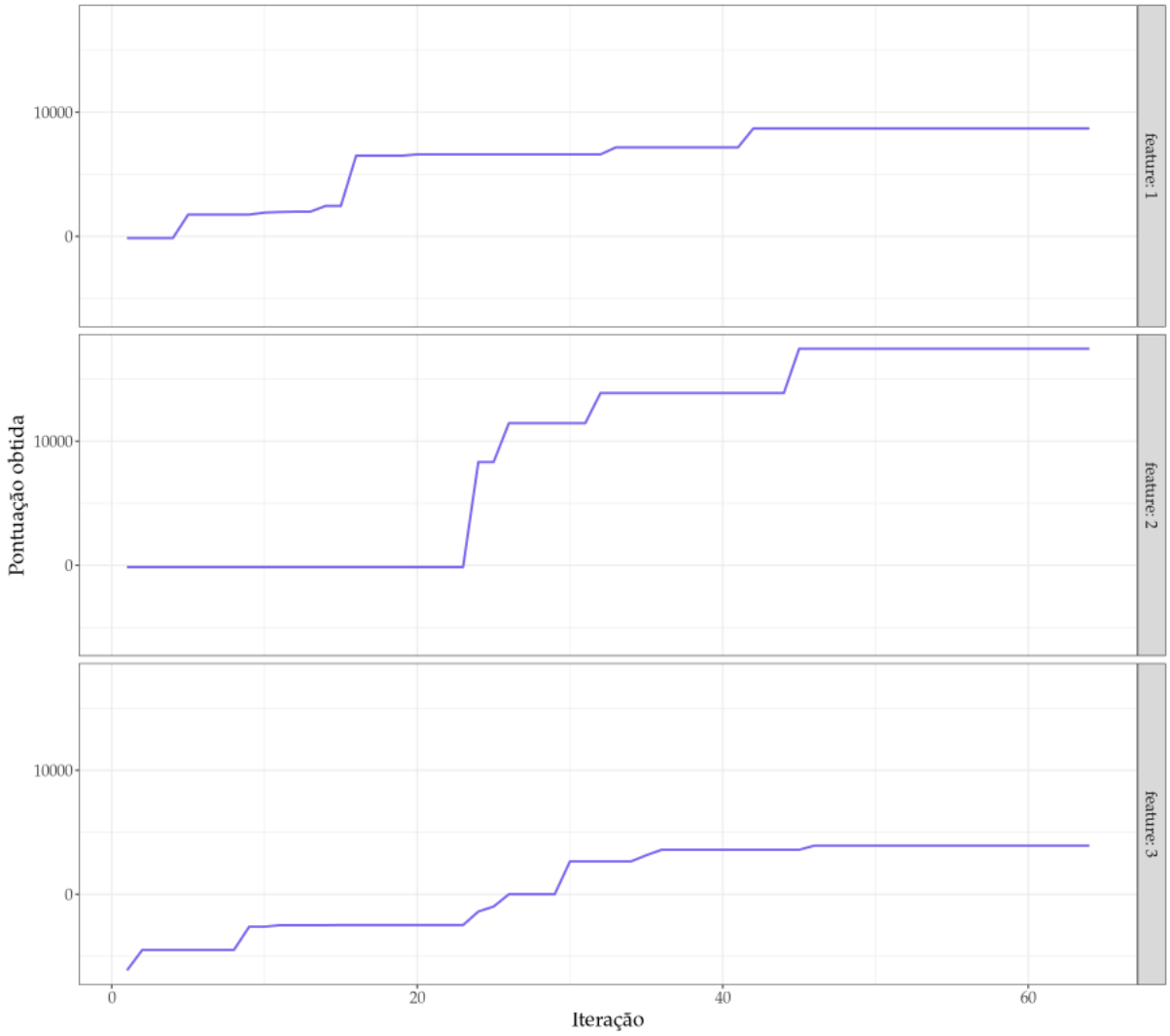

Figura 16: Visualização da pontuação obtida pelo HC nos 3 conjuntos de *features* com o treinamento realizado com só uma pista e oponente, com porcentagem de perturbação fixada em, respectivamente, 0.7, 0.5 e 0.2 para cada feature

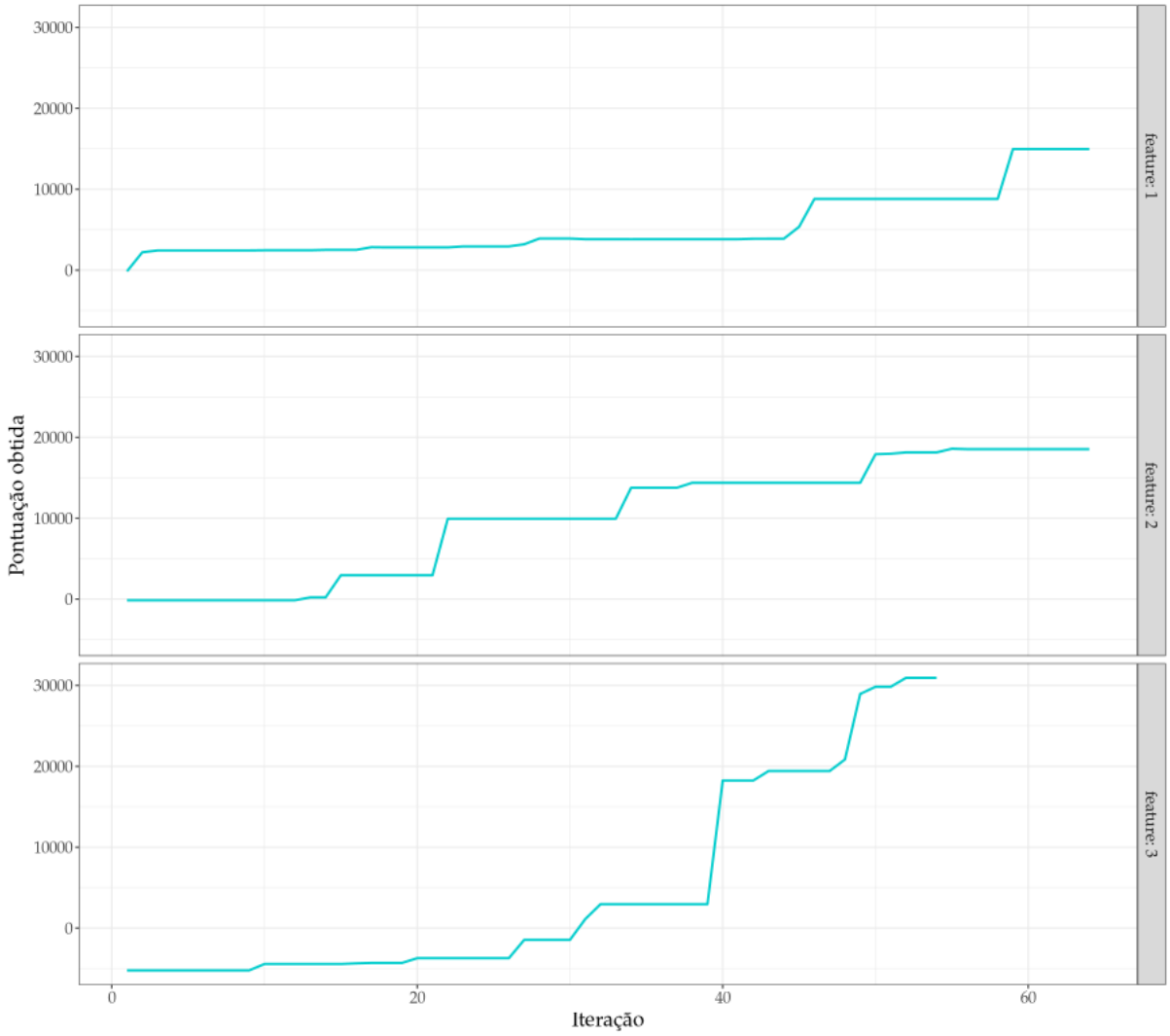

Figura 17: Visualização da pontuação obtida pelo GA nos 3 conjunto de *features* com o treinamento realizado com só uma pista e oponente, com mutação e eletismo fixados em 0.25 e 0.2 e com a população em 60 para a primeira e terceira faceta e em 40 para a segunda

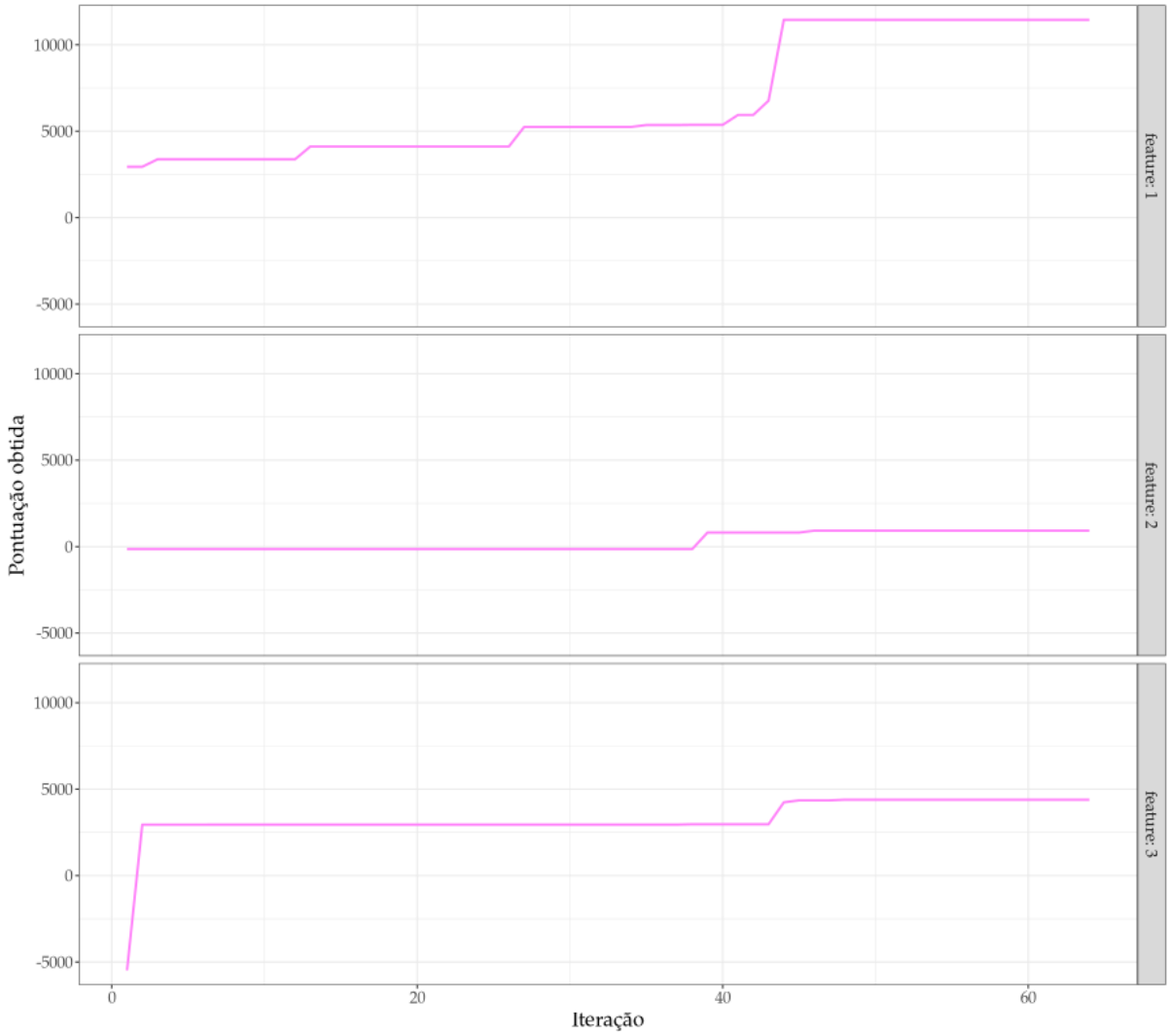

Figura 18: Visualização da pontuação obtida pelo/CMA-ES/nos 3 conjunto de *features* com o treinamento realizado com só uma pista e oponente