Uma abordagem para geração automática de dados de teste utilizando algoritmos evolutivos para software controlador de veículos autônomos

Vânia de Oliveira Neves[1], Márcio Eduardo Delamaro[1], Paulo Cesar Masiero[1]

Depto de Sistemas de Computação - ICMC
Universidade de São Paulo - São Carlos, SP - Brasil
{vaniaon,delamaro,masiero}@icmc.usp.br

Resumo Veículos autônomos são um tipo de sistema robótico móvel que têm como principal objetivo mover e fazer diversas manobras, como ultrapassar outro veículo, estacionar e obedecer a regras de trânsito sem a presença de um condutor, ou seja, autonomamente. São sistemas críticos e, como tais, devem ser suficientemente testados. Pesquisas envolvendo o teste estrutural do software controlador de veículos autônomos vêm sendo realizadas e, neste artigo, é apresentado um algoritmo evolutivo para geração automática de dados de entrada para esse tipo de sistema. Também é apresentado um estudo envolvendo a geração desses dados de entrada a partir de logs coletados em cinco testes de campo de um veículo autônomo desenvolvido no ICMC-USP. O estudo mostrou-se promissor uma vez que foi possível melhorar a cobertura obtida previamente pelos logs obtidos nos testes de campo.

1 Introdução

Veículos autônomos vêm ganhando muita importância nos últimos anos. Para promover pesquisas na área, a Defense Advanced Research Projects Agency (DARPA) realizou três competições: as de 2004 e 2005 tinham foco em veículos capazes de atravessar o deserto com capacidade de seguir uma rota e desviar de obstáculos. Na terceira competição, realizada em 2007, os veículos deveriam ser capazes de navegar de uma maneira segura em um ambiente urbano interagindo tanto com outros veículos autônomos, quanto com veículos guiados por humanos. No entanto, à medida que essa área avança, outras questões importantes surgem como por exemplo, como saber se esses veículos foram suficientemente testados.

Em geral, o processo de desenvolvimento do software controlador de um veículo autônomo segue um processo iterativo em que, após a implementação de suas funcionalidades, os requisitos são validados por meio de simulação. Em seguida, são realizados testes de campo com o veículo real em que é possível gravar os dados de entrada recebidos para posterior reprodução e análise offline. Esse processo se repete até que a tecnologia seja aceita[15]. No entanto, os testes realizados são basicamente funcionais e não levam em consideração nenhuma informação a respeito da estrutura do código fonte. Ainda que o veículo tenha passado por vários testes de campo e se comportado da maneira esperada, é possível que trechos críticos do software nunca tenham sido executados.

Nesse contexto, o objetivo deste trabalho é propor uma abordagem para geração de dados de teste utilizando algoritmos evolutivos e tendo como função objetivo (ou *fitness*) os requisitos do teste estrutural. Nesta abordagem, a geração é realizada após a execução de um teste de campo e utiliza as entradas gravadas durante essa execução. Embora existam várias estratégias consolidadas para geração automática de dados de teste baseada em busca, a maioria delas considera variáveis de entradas mais simples e comuns, como números e vetores. No entanto, em alguns domínios, como no de sistemas embarcados, há também a ocorrência de outros tipos mais complexos de estruturas de dados. Nos veículos autônomos, um tipo de entrada recebida são as nuvens de pontos, que são estruturas organizadas como matrizes complexas obtidas a partir do processamento das imagens recebidas por uma câmera acoplada a eles.

Este artigo está organizado da seguinte forma: os trabalhos relacionados são apresentados na Seção 2; as adaptações necessárias para a utilização de algoritmos evolutivos no contexto de veículos autônomos são apresentadas na Seção 3; na Seção 4 é apresentado o algoritmo proposto e uma descrição dele; na Seção 5 é apresentado um estudo exploratório com o objetivo de realizar uma validação inicial da abordagem proposta e, por fim, a Seção 6 contém as conclusões e trabalhos futuros.

2 Trabalhos relacionados

Os algoritmos genéticos começaram a ser usados na área de teste para geração de casos de teste nos anos 90. Alguns artigos, como os de Jones et al. [5] e Pargas et al. [14] foram pioneiros nessa área. De um modo geral eles usavam uma população inicial gerada aleatoriamente e desenvolveram o conceito de função de *fitness* com base em critérios de teste estruturais (nós, arestas, etc). Além disso, esses primeiros trabalhos focavam em estruturas de dados tradicionais, como números inteiros e flutuantes, escalares e vetores. Posteriormente, outras estruturas de dados mais complexas como cadeias de caracteres [6] e estruturas de dados do domínio de telecomunicações [2], foram investigadas.

Há alguns anos McMinn [7] realizou uma revisão bibliográfica dos trabalhos na área de geração de dados de teste baseada em busca. Entre as técnicas dessa área estão os algoritmos evolutivos, que usam evolução simulada como estratégia de busca para encontrar candidatos que melhor satisfazem a função de *fitness*, utilizando operadores inspirados pela geração genética e natural. Algoritmos Genéticos são a forma mais conhecidas de Algoritmos Evolutivos. O Algoritmo 1 apresenta uma descrição em alto nível desse algoritmo, retirada de McMinn [7].

Algorithm 1 Descrição em alto nível de um Algoritmo Genético [7]

1: Gerar aleatoriamente ou semear a população inicial P
2: **Repita**
3: Avaliar a função de fitness de cada indivíduo em P
4: Selecionar os pais a partir de P de acordo com o mecanismo de seleção
5: Recombinar os pais para formar o novo offspring
6: Construir a nova população P' a partir dos pais e do offspring
7: Mutar P' $P \leftarrow P'$
8: **Até** Atingir a condição de parada

Há poucos estudos que discutem a geração de casos de teste para algoritmos de busca no domínio de software embarcado [13, 2, 1]. Com relação ao teste estrutural, Doganay et al. [2] apresentam um estudo de caso de uma aplicação de técnicas de teste de software baseadas em busca para um software embarcado industrial no domínio de telecomunicações, com o objetivo de gerar dados de entrada no nível de teste de unidade, considerando os critérios de cobertura todos-nós e todas-arestas do grafo de fluxo de controle. Com o estudo de caso conduzido verificou-se que a abordagem aumentou a cobertura mas, no entanto, a geração foi tão efetiva quanto a geração aleatória. Para a técnica de teste funcional, Bühler and Wegener [1] apresentam uma abordagem para gerar casos de teste utilizando técnicas evolutivas para um sistema de estacionamento autônomo. Eles propõem e avaliam abordagens para definir a função de *fitness* nesse domínio. Nguyen et al. [13] utilizaram algoritmos evolutivos para gerar automaticamente casos de teste para agentes autônomos. As funções de *fitness* foram definidas a partir dos requisitos funcionais de alto nível do sistema.

3 Adaptação do algoritmo genético para a área de veículos autônomos

Ao realizar o teste de campo, todos os dados de entrada recebidos são armazenados em um arquivo de log que podem ser usados como entrada para futura simulação e reexecução offline. Dentre as entradas armazenadas nesse log estão as nuvens de pontos, que são geradas a partir da transformação das imagens recebidas pela câmera estéreo do veículo. Ela é organizada em uma estrutura de matriz e contém informações das coordenadas (x, y e z) e de cores (r, g, b) para cada ponto da imagem, além de outros dados. Para que o programa possa ser executado, as nuvens de pontos devem ser reproduzidas por meio do arquivo de logs. Assim, o algoritmo genético geral para geração de casos de testes apresentado na Seção 2 foi adaptado para gerar novos logs que contêm nuvens de pontos com o objetivo de serem utilizadas no contexto de veículos autônomos. Para essa adaptação, os indivíduos são logs (ou trechos de logs) que contêm nuvens de pontos. Além disso, três aspectos foram considerados: a geração da população inicial, a estratégia para o operador de combinação e a estratégia para o operador de mutação. Os três são discutidos na próxima seção.

O algoritmo deve ser genérico e poder ser utilizado com diferentes estratégias de combinação e mutação. Neste artigo o algoritmo é apresentado e avaliado com a duas estratégias descritas abaixo, mas outros estudos com diferentes estratégias estão sendo realizados.

3.1 Seleção da População Inicial

Normalmente, a população é gerada aleatoriamente, mas estudos demonstraram que o uso de uma população inicial adequada pode produzir melhores resultados [4]. Por isso, decidiu-se por selecionar um conjunto inicial reduzido de indivíduos e que esses fossem representativos a fim de levarem a uma grande cobertura de execução do programa.

A população inicial foi gerada a partir da intuição física de como o programa se comporta quando executado: de um modo geral, em cada ciclo de leitura da nuvem

de pontos, se não há obstáculos no caminho que precisem de desvios e mudanças de velocidade, é natural que o programa execute os mesmos comandos computacionais e que os testes de condições não mudem. Neves et al. [12] apresentou uma abordagem para particionar os logs em trechos (ou subsequências) em que cada trecho contém um número fixo (parametrizável) de nuvens de ponto. Nesse estudo, a cobertura de cada um desses trechos foi calculada e, analisando os resultados obtidos, notou-se que há trechos que têm uma contribuição maior para a cobertura. Esses trechos foram interpretados como aqueles em que houve alguma mudança nas condições do percurso que levaram o programa a percorrer outros caminhos em seu fluxo de execução. A eles foi dado o nome de trechos discriminatórios.

Conforme a intuição inicial, eles ocorrem em pequeno número e podem ser bons candidatos para gerar a população inicial buscada. Com base neles foram criadas e testadas sete heurísticas: as heurísticas H1 e H2 são focadas em maximizar a cobertura de um método para um critério de teste; as heurísticas H3 e H4 são uma composição das duas primeiras com os trechos inicial e final, considerando que nestes pontos podem ocorrer computações diferentes do meio do percurso; a quinta heurística (H5) junta os trechos discriminatórias do critério todos-nós para todos os métodos sob teste, com o objetivo de maximizar a cobertura de todos os métodos envolvidos no teste; a heurística H6 é semelhante à H5, mas para o critério todas-arestas; e a heurística H7 seleciona os trechos discriminatórios de maneira aleatória. No estudo, que está descrito detalhadamente em Neves et al. [12], as heurísticas que envolveram a seleção dos trechos discriminatórios do critério todas-arestas (H2 e H6) obtiveram os melhores resultados.

A população inicial, portanto, é gerada a partir do conjunto de trechos selecionados automaticamente de acordo com uma determinada heurística definida em Neves et al. [12]. Se o número de indivíduos dessa população for menor que o tamanho da população desejada, são incluídos posteriormente pelo algoritmo de geração novos trechos gerados a partir dos operadores genéticos de combinação e mutação desses indivíduos até obter o tamanho da população inicial desejada.

3.2 Combinação dos pais: metades verticais

A estratégia de combinação dos pais para gerar novos filhos é definida utilizando dois trechos consecutivos T_1 e T_2 da população. A geração combina duas metades verticais de uma nuvem de pontos np_{i_j} de T_1 com a nuvem de pontos np_{i+1_j} de T_2, gerando duas novas nuvens de pontos. A forma de combinação é ilustrada na Figura 1.

3.3 Estratégia de mutação: inserção de obstáculos

Esta estratégia de mutação insere um ou mais objetos simulados em uma nuvem de pontos np gerando uma nova nuvem np'. Os objetos são retangulares e podem simular postes, árvores, carros, etc. Esta estratégia deve levar em conta quatro decisões a serem tomadas:

1. **A quantidade de obstáculos inseridos:** que é calculada aleatoriamente pelo algoritmo;

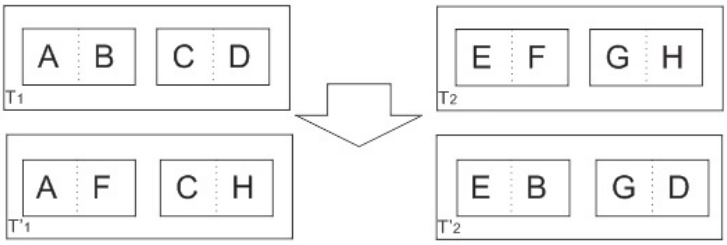

Figura 1. Esquema de combinação de nuvens de pontos

2. **Posição dos obstáculos:** calculada aleatoriamente pelo algoritmo considerando uma zona delimitadora para a geração aleatória da posição inicial do objeto. A implementação escolhe aleatoriamente uma constante k que pode variar de 1% a 20% da menor medida da matriz e, então, desconsidera $k\%$ das primeiras linhas e $k\%$ das últimas colunas. Considera-se o ponto do canto superior esquerdo para definir se o objeto está ou não dentro da zona permitida.
3. **Valor da constante z:** calculado aleatoriamente pelo algoritmo. Geralmente, a altura dos pontos (coordenada z) é usada pelos programas que controlam veículos autônomos para determinar a altura do obstáculo. Por isso, essa coordenada é aumentada para todos os pontos dentro dos limites do objeto inserido usando uma constante definida aleatoriamente.
4. **Tamanho máximo dos objetos:** parâmetro que corresponde ao tamanho máximo, em percentual, em relação ao tamanho da matriz de nuvem de pontos. A partir desse percentual, uma constante é escolhida aleatoriamente entre 1% e o valor informado como parâmetro. O tamanho então é calculado considerando o percentual dessas constantes aplicados às medidas de tl (tamanho das linhas) e tc (tamanho das colunas). Se um objeto ultrapassar as bordas da matriz, esses valores serão desconsiderados e, como consequência, o objeto será menor do que o calculado. A Figura 2 apresenta um exemplo de um objeto inserido. A posição aleatória escolhida foi o ponto A da figura. O tamanho calculado do objeto foi ACFD e a parte BCFE está fora do intervalo da matriz de nuvens de pontos. Assim, o tamanho considerado foi ABED.

A cada mutação realizada, esses valores são recalculados. Portanto, a quantidade, tamanho e a posição dos objetos podem variar para cada novo indivíduo.

4 Algoritmo evolutivo para geração de dados de teste para software controlador de veículos autônomos

Uma adaptação do algoritmo evolutivo para o contexto de veículos autônomos que utilizam trechos que contêm nuvens de pontos como dados de entrada é apresentada no Algoritmo 2. Os trechos são avaliados executando o programa sob teste por meio de simulação e utilizando-os como entrada. No final da execução do programa, obtem-se o conjunto de dados de teste gerado bem como uma lista dos requisitos de teste cobertos

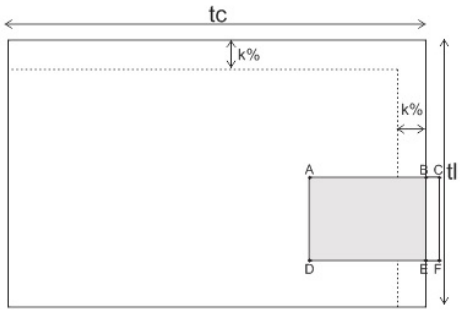

Figura 2. Exemplo de inserção de um obstáculo que ultrapasse a borda da matriz

por eles e o percentual de cobertura. Assim como nos algoritmos genéticos tradicionais, a avaliação da função de *fitness* é utilizada para selecionar os pais da próxima geração e os operadores genéticos de combinação e mutação são utilizados para gerar a nova população de dados de teste. A avaliação da função de *fitness* depende do número de requisitos cobertos: ela é maior para o trecho que cobriu o maior número de requisitos. Conforme explicado adiante, o cálculo da função de *fitness* é diferente para o conjunto utilizado como população inicial dos demais indivíduos gerados.

O algoritmo utiliza como entrada os seguintes dados: *versao*, que é a versão de um programa a ser testado; *logTC*, um log gerado a partir do teste de campo; *numNP*, que corresponde ao número de nuvens de pontos que cada trecho particionado de *logTC* deve conter; *fitness*, critério de teste estrutural a ser considerado: todos-nós ou todas-arestas; *tamPopulacao*, o tamanho da população de dados de teste a ser considerado na geração; *iteracoes*, o número de iterações a ser considerado como um dos critérios de parada do algoritmo; *metodosSobTeste*: lista dos métodos da versão sob teste que deverão ser considerados na geração; *heuristica*, heurística a ser utilizada na seleção do conjunto inicial de trechos (*conjtoInicial*); *opCombinacao*, operador de combinação a ser utilizado na combinação dos pais para a geração de novos filhos; *opMutacao*, operador de mutação a ser utilizado para geração de novos filhos; e, *taxaMutacao*, valor da taxa de mutação a ser usado para gerar ou não um novo trecho mutado.

Adicionalmente, o algoritmo tem como variáveis locais: *trechosPart*, que armazena a lista de trechos particionado de *logTC*; *iteracaoAtual*, variável que armazena a quantidade de iterações que já foram realizadas; *mutacao*, valor em porcentagem calculado aleatoriamente para decidir se irá ou não gerar trechos mutados; *reqCobertos*, conjunto que armazena os requisitos que já foram cobertos pelos trechos gerados; *reqCobertos-ConjtoInicial*, conjunto que armazena os requisitos cobertos apenas por *conjtoInicial*; *populacaoCorrente*, lista que armazena os trechos referente à população corrente; *conjtoInicial*, lista de trechos que foram selecionados pela heurística (*heuristica*) desejada; *filhos*, lista de trechos que foram gerados na iteração atual (*iteracaoAtual*); *cobertura*, percentual de cobertura atingido pela população.

O algoritmo começa inicializando e configurando as variáves locais e de entrada (linhas 1-11). *logTC* é particionado em trechos contendo *numNP* nuvens de pontos cada

e os trechos particionados são armazenados na lista *trechosPart*. Em seguida, são selecionados os trechos discriminatórios de acordo com *heuristica* para os métodos na lista *metodosSobTeste*. Esses trechos são armazenados na lista *conjtoInicial* e são os indivíduos da população inicial do algoritmo genético proposto.

A fim de obter uma lista de requisitos de teste de acordo com a função de *fitness*, o Grafo de Fluxo de Controle (GFC) é calculado (linha 3) e o número desses requisitos de teste é armazenado na variável *numRequisitos*. Para descobrir quais requisitos de teste foram cobertos por cada trecho, a *versao* do programa sob teste deve ser executada recebendo cada um dos trechos como entrada. Isso é realizado pelo procedimento *simular()*. Essa lista de requisitos cobertos é atribuída à variável *reqCobertos-ConjtoInicial*. Com isso, o valor da função objetivo de cada elemento de *conjtoInicial* é calculado segundo a fórmula: $numReqCobertosTrecho/numRequisitos$, em que $numReqCobertosTrecho$ representa o número de requisitos cobertos por um dado trecho do conjunto inicial. Também na etapa de inicialização é calculada a cobertura inicial obtida por *conjtoInicial*, a lista de trechos de *conjtoInicial* é atribuída à variável *populacaoCorrente* e o valor zero é atribuído à variável *iteracaoAtual*.

O próximo passo é a geração de dados de teste propriamente dita (linhas 12-22). O laço **Enquanto** é executado até atingir 100% de cobertura ou até que o número de iterações realizadas seja igual ao valor de *iteracoes*. O primeiro passo dessa fase é gerar os filhos da população corrente utilizando o operador de combinação informado em *opCombinacao* e atribuindo esse resultado à variável *filhos*. Inicialmente, apenas o operador de metades verticais foi definido (ver Seção 3.2), no entanto, o algoritmo já considera que outros operadores possam ser utilizados no futuro. Em seguida, os filhos gerados são passados como parâmetros para o procedimento *realizaMutacao*. Nesse procedimento, para cada um dos indivíduos em filhos é gerado um valor aleatório *mutacao* a fim de compará-lo com a variável de entrada *taxaMutacao* com o objetivo de decidir se será aplicado ou não o operador de mutação para um dado indivíduo. Em caso positivo, o indivíduo será mutado utilizando o operador de mutação informado em *opMutacao* e substituindo o indivíduo original na lista de *filhos*. Inicialmente, apenas o operador de inserção de obstáculos foi definido (ver Seção 3.3), no entanto, o algoritmo já considera que outros operadores possam ser utilizados no futuro.

Como o número de trechos selecionados a partir da heurística (*conjtoInicial*) pode ser menor que o tamanho da população inicial informada em *tamPopulacao*, o algoritmo verifica se o número de trechos de *populacaoCorrente* + número de trechos gerados em *filhos* atingiu o tamanho desejado. Em caso positivo, o procedimento *selecionarPais* é invocado com a finalidade de descartar os indivíduos que menos contribuíram para a cobertura dos requisitos de teste, respeitando o valor de *tamPopulacao*; a variável *iteracaoAtual* é incrementada e o valor de cobertura atingido pelos trechos de *populacaoCorrente* é atualizada. Em caso negativo, os elementos de *filhos* são adicionados à lista de trechos de *populacaoCorrente* e o laço se repete. Nota-se, neste caso, que o valor da variável *iteracaoAtual* não sofre nenhuma alteração.

Para fazer a seleção dos indivíduos que permanecerão na próxima iteração, o procedimento *selecionarPais* primeiramente calcula os requisitos cobertos pelos trechos pertencentes à lista *filhos*, invocando o método *simular* para cada um deles. A etapa de simulação não é necessária para os indíviduos que estão na lista *populaçãoCorrente* uma

vez que já foi realizada em iterações anteriores. A partir da lista de requisitos cobertos por cada trecho, é possível calcular então a função de *fitness* para cada um deles segundo a fórmula: $numReqCobertosFilho/numRequisitos$, onde $numReqCobertosFilho$ representa o número de requisitos cobertos por um dado trecho da lista *filhos*, desconsiderando os requisitos que já haviam sido cobertos pelos trechos em *conjtoInicial*. Vale ressaltar que esse cálculo é diferente do cálculo realizado para calcular a função *fitness* do conjunto inicial e funciona como um mecanismo de elitismo para dificultar que os trechos de *conjtoInicial* sejam descartados durante as iterações. Essa priorização é feita porque os trechos de *conjtoInicial* são indivíduos que foram selecionados porque possuiam grandes chances de aumentar a cobertura e, portanto, não seria vantajoso retirá-los da população. O próximo passo é adicionar os trechos de *filhos* à *populacaoCorrente* para que se possa ordenar de acordo com os valores da função de *fitness*. Com isso, os trechos que obtiveram os menores valores da função de *fitness* são removidos da *populacaoCorrente*, de tal forma que o número de trechos de *populacaoCorrente* seja igual a *tamPopulacao*.

5 Validação da abordagem proposta

Para validar a abordagem proposta, foi conduzido um estudo utilizando o software controlador do veículo autônomo CaRINA [3], desenvolvido pelo Laboratório de Robótica Móvel do ICMC/USP, e os logs obtidos a partir de testes de campo com esse veículo. O módulo utilizado para os testes foi o de navegação autônoma utilizando visão estéreo e desvio de obstáculos [9, 8]. O algoritmo foi implementado na ferramenta desenvolvida por Neves et al. [10, 11], sendo então estendida para oferecer suporte à estratégia proposta.

5.1 Módulo para navegação autônoma e desvio de obstáculos

O módulo de navegação autônoma utilizado permite dirigir um veículo autônomo em uma região urbana para chegar a um determinado ponto definido por uma coordenada de GPS, evitando colisões. Ele é dotado de uma câmera estéreo que fornece duas imagens, as quais são processadas por um método semi-estéreo global para produzir um mapa de disparidade. Esse mapa é então convertido para uma nuvem de pontos em 3D com base em parâmetros da câmera. A orientação da câmera em relação ao solo é estimada pelo método RANSAC (*Random Sample Consensus*) e é usado um método de detecção de obstáculos que classifica os pontos com base em elevações e diferenças relativas de altura. Essas informações são usadas como entradas para o *Vector Field Histogram* (VFH) que é usado para guiar o veículo até o ponto de chegada [9]. O diagrama de classes desse programa é mostrado na Figura 3.

O programa foi implementado utilizando as funcionalidades do framework ROS. Dessa forma, as leituras tanto da câmera estéreo quanto dos outros dados de entrada do programa são publicadas pelo ROS e recebidas pelo programa controlador por meio do método $main()$. Esse método, por sua vez, envia esses dados de entrada para a classe $cObstacleAvoidance$, que é responsável pelos processamentos descritos no parágrafo acima, para então tomar a decisão sobre a atuação do veículo: acelerar, desacelerar,

Algorithm 2 Algoritmo Genético adaptado para gerar casos de testes para programas controladores de veículos autônomos

Entrada:

fitness: critério de cobertura a ser utilizado: nós ou arestas
heuristica: heurística a ser considerada na seleção do conjunto de trechos inicial
iteracoes: número de iterações a ser considerada como critério de parada
logTC: log gerado a partir do teste de campo
metodosSobTeste: lista de métodos a serem considerados na função de fitness
numNP: número de nuvens de pontos em cada trecho
opCombinacao: operador de combinação a ser utilizado
opMutacao: operador de mutação a ser utilizado
tamPopulacao: tamanho da população
taxaMutacao: valor da taxa de mutação
versao: versão do programa sob teste

Variáveis:

cobertura: valor de cobertura
conjtoInicial: lista contendo os trechos discriminatórios de *logTC*
filhos: lista contendo os indivíduos gerados por combinação na *iteracaoAtual*
iteracaoAtual: número de iterações que já foram realizadas
mutacao: valor da taxa de mutação calculada aleatoriamente
numRequisitos: número de requisitos de teste
populacaoCorrente, *conjtoInicial*, *filhos*: lista de trechos
reqCobertos: conjunto dos requisitos de teste cobertos pelos dados gerados
reqCobertosConjtoInicial: conjunto dos requisitos de teste que foram cobertos pelos trechos em *conjtoInicial*
trechosPart: lista contendo trechos de *logTC* particionado

Funções e procedimentos:

gerarConjuntoInicial(): função que retorna uma lista com os trechos discriminatórios de *logTC*
particionarLog(): procedimento responsável por particionar *logTC* em trechos contendo *numNP* nuvens de pontos
simular(): procedimento que executa a *versao* do programa sob teste tendo como entrada um determinado trecho
combinarPopulacao(): procedimento que gera novos indivíduos combinando-os dois a dois de acordo com *opCombinacao*
realizarMutacao(): procedimento que calcula um valor aleatório para cada indivíduo e realiza a mutação se esse valor for maior ou igual que *taxaMutacao*
selecionarPais(): procedimento que faz a seleção dos pais para a próxima iteração

/**** Inicialização e configuração ****/

trechosPart ← *particionarLog(numNP)*
conjtoInicial ← *gerarConjuntoInicial(trechosPart, heuristica, metodosSobTeste)*
Calcular o GFC dos métodos sob teste
Calcular requisitos de teste e atribuir a quantidade de requisitos à variável *numRequisitos*
simular(conjtoInicial)
Computar os requisitos cobertos de *conjtoInicial* e adicioná-los em *reqCobertos*
Computar o valor de fitness dos elementos de *conjtoInicial*, segundo a fórmula $numReqCobertosTrecho/numRequisitos$
Armazenar os requisitos cobertos por *conjtoInicial* em *requisitosCobCnjtoInicial*
cobertura ← $reqCobertos.size()/numRequisitos$
populacaoCorrente ← *conjtoInicial*
iteracaoAtual ← 0

/**** Geração dos dados de teste ****/

Enquanto iteraçãoAtual < iteracoes **E** cobertura != 1 **faça**
    filhos ← *combinarPopulacao*(*populacaoCorrente*, *opCombinacao*)
    filhos ← *realizarMutacao*(*filhos*, *opMutacao*, *taxaMutacao*)
    Se ((filhos.size() + populacaoCorrente.size()) > tamPopulacao) **então**
        populacaoCorrente ← *selecionarPais*(*populacaoCorrente*, *filhos*)
        iteracaoAtual++
        cobertura = $reqCobertos.size()/numRequisitos$
    Senão
        populacaoCorrente ← *populacaoCorrente* + *filhos*
    Fim Se
Fim Enquanto

Procedimento SELECIONARPAIS($populacaoCorrente$, $filhos$)
    simular(filhos)
    Para *filho* em *filhos* **faça**
        filho.reqCobertos ← *filho.reqCobertos - reqCobertosConjtoInicial*
        fitness ← *filho.reqCobertos.size()*/*numRequisitos*
        populacaoCorrente ← *populacaoCorrente* + *filhos*
        Ordenar *populacaoCorrente*, considerando a função de fitness de cada elemento
        Retirar os (*populacaoCorrente.size() - tamPopulacao*) indivíduos de *populacaoCorrente*
    Fim Para
Fim Procedimento

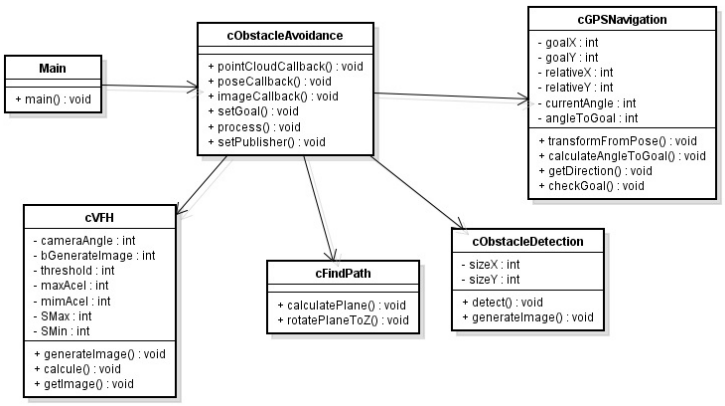

Figura 3. Diagrama de Classes

frenar, mudar o ângulo das rodas etc. Os dois métodos principais do programa, responsáveis por esses cálculos e decisões são $process()$, da classe $cObstacleAvoidance$, e $calcule()$, da classe $cVFH$. O primeiro tem 155 linhas de código sem comentários e o segundo 153. O estudo exploratório e as análises apresentadas neste artigo foram baseados nesses dois métodos, pois os demais são mais simples e pequenos, atingindo facilmente altos níveis de cobertura de nós e arestas.

5.2 Validação da abordagem proposta

Um estudo foi conduzido com o objetivo de validar o algoritmo para geração de dados de teste proposto. Para isso, foram realizados cinco testes de campo com o veículo autônomo a fim de coletar logs com as entradas recebidas durante os testes. Em seguida, esses logs foram utilizados como entrada da variável *logTC* para aplicação do algoritmo. Os outros valores das variáveis de entrada utilizadas foram: *versao*: a última versão do programa descrito na Seção 5.1; *numNP*: 5; *fitness*: todas-arestas; *tamPopulacao*: 25; *iteracoes*: 10; *metodosSobTeste*: $process()$, $calcule()$; *heuristica*: H2, H6; *opCombinacao*: metades verticais; *opMutacao*: inserção de obstáculos; *taxaMutacao*: 20. Com isso, inicialmente, para cada uma das execuções, os logs foram particionados em trechos contendo 5 nuvens de pontos cada e as heurísticas informadas como parâmetro foram aplicadas nesses logs a fim de se obter um conjunto de trechos discriminatórios. Em seguida, a cobertura obtida por esses trechos foi calculada para cada log. As coberturas alcançadas inicialmente são apresentadas na primeira parte da Tabela 1. Nessa tabela, a coluna *ID* indica o ID da execução; a coluna *Log* indica qual o log que foi executado; a coluna *MetodosSobTeste* lista quais os métodos sob teste que foram considerados; a coluna *Heuristica* indica a heurística considerada; a coluna *Cob. Ini - Nós* indica a porcentagem de cobertura obtida para o critério todos-nós; a coluna *Cob. Ini - Arestas* indica a porcentagem de cobertura obtida para o critério todas-arestas. Quando mais de um método é informado em *metodosSobTeste*, as coberturas apresentadas nas colunas *Cob. Ini - Nós* e *Cob. Ini - Arestas* são apresentadas indicando a sequência dos métodos

Tabela 1. Cobertura inicial e cobertura alcançada com os logs gerados

ID	Log	MetodosSobTeste	Heurística	Cob. Ini. - Nós	Cob. Ini. - Arestas	Cob. Nós	Cob. Arestas
1	1	<process>	H2	63%	48%	71%	59%
2	1	<calcule>	H2	91%	74%	91%	74%
3	1	<process, calcule>	H6	<63%, 92%>	<48%, 75%>	<71%, 92%>	<59%, 75%>
4	2	<process>	H2	74%	58%	79%	71%
5	2	<calcule>	H2	93%	77%	93%	77%
6	2	<process, calcule>	H6	<74%, 93%>	<58%, 78%>	<79%, 93%>	<72%, 78%>
7	3	<process>	H2	63%	48%	70%	59%
8	3	<calcule>	H2	91%	73%	91%	73%
9	3	<process, calcule>	H6	<63%, 91%>	<48%, 73%>	<70%, 91%>	<59%, 74%>
10	4	<process>	H2	74%	58%	79%	72%
11	4	<calcule>	H2	90%	73%	90%	73%
12	4	<process, calcule>	H6	<74%, 93%>	<59%, 77%>	<79%, 93%>	<72%, 77%>
13	5	<process>	H2	63%	48%	71%	59%
14	5	<calcule>	H2	92%	75%	92%	75%
15	5	<process, calcule>	H6	<63%, 92%>	<48%, 75%>	<70%, 92%>	<59%, 75%>

na coluna *metodosSobTeste*. Por exemplo, para a execução de ID 3, a cobertura inicial para o critério todos-nós do método $process$ foi de 63% e para o método $calcule$ foi de 92%.

O algoritmo então prossegue a execução considerando esses valores para gerar novos trechos do log. Após o término do algoritmo, o conjunto desses gerados em cada execução foi simulado a fim de computar a cobertura obtida. Esse resultado também pode ser visto na Tabela 1 em que a coluna *Cob. Nós* indica qual foi o percentual de cobertura atingido pela execução dos dados de teste gerados para o critério todos-nós de cada método informado; e a coluna *Cob. Arestas* indica qual foi o percentual de cobertura atingido pela execução dos dados de teste gerados para o critério todas-arestas de cada método informado. De forma análoga à cobertura inicial, quando mais de um método é informado em *metodosSobTeste*, as coberturas apresentadas nas colunas *Cob. Nós* e *Cob. Arestas* são apresentadas indicando a sequência dos métodos na coluna *metodosSobTeste*. Assim, para a execução de ID 3, a cobertura obtida para o critério todas-arestas do método $process$ foi de 59% e para o método $calcule$ foi de 75%.

Comparando as colunas *Cob.Ini - Nós* e *Cob. Ini - Arestas* com as colunas *Cob. Nós* e *Cob.Arestas* da Tabela 1, pode-se notar que houve uma boa melhora na cobertura. Ou seja, o algoritmo conseguiu gerar trechos capazes de cobrir requisitos de teste que ainda não haviam sido cobertos pelos trechos utilizados como conjunto inicial. Para o log 2, por exemplo, a cobertura para o critério todas-arestas passou de 58% para 71%. O resultado foi satisfatório uma vez que tanto o número de iterações quanto o tamanho da população utilizados foram pequenos. Além disso, para alguns logs a cobertura já estava muito alta o que torna mais difícil encontrar novos dados de teste que cubram os requisitos ainda não cobertos.

6 Conclusões e Trabalhos Futuros

Este artigo apresentou uma abordagem para geração de dados de teste utilizando algoritmos evolutivos, mais especificamente algoritmos genéticos, e considerando critérios do teste estrutural no contexto de veículos autônomos. Um algoritmo genético adaptado para o contexto de veículos autônomos foi apresentado. Entre as adaptações para

esse algoritmo estão a definição de uma estratégia para seleção da população inicial e a definição de operadores genéticos de combinação e mutação.

Além disso, a ferramenta desenvolvida por Neves et al. [10, 11] foi estendida para oferecer suporte à estratégia proposta, possibilitando que um estudo fosse realizado. Esse estudo utilizou cinco logs reais obtidos em testes de campo de um veículo autônomo e, mesmo utilizando tamanho da população e número de iterações pequenos, foi possível melhorar a cobertura obtida previamente.

A aplicação da abordagem possibilita, por meio de análise dos trechos gerados, pensar em cenários de teste reais para serem reproduzidos em testes de campo de forma a garantir a mesma cobertura obtida na execução offline. Além disso, esse conjunto gerado poderia ser utilizado em testes de regressão.

Como trabalhos futuros pretende-se definir novas estratégias de combinação e de mutação da nuvens de pontos. Como a abordagem pode gerar vários dados de teste que não poderiam ser reproduzidos na vida real, um outro trabalho futuro seria a definição de operadores genéticos que gerassem trechos que fossem capazes de serem reproduzidos em cenários reais.

Agradecimentos

Os autores agradecem à FAPESP, CAPES, CNPq e INCT-SEC pelo apoio financeiro.

Referências Bibliográficas

[1] Bühler, O., Wegener, J.: Automatic testing of an autonomous parking system using evolutionary computation. Tech. rep., SAE Technical Paper (2004)

[2] Doganay, K., Eldh, S., Afzal, W., Bohlin, M.: Search-based testing for embedded telecom software with complex input structures. In: Merayo, M., de Oca, E. (eds.) Testing Software and Systems, Lecture Notes in Computer Science, vol. 8763, pp. 205–210. Springer Berlin Heidelberg (2014)

[3] Fernandes, L.C., Souza, J.R., Pessin, G., Shinzato, P.Y., Sales, D.O., Mendes, C.C.T., Prado, M., Klaser, R.L., Magalhães, A.C., Hata, A.Y., Pigatto, D.F., Branco, K.R.L.J.C., Jr., V.G., Osório, F.S., Wolf, D.F.: Carina intelligent robotic car: Architectural design and applications. Journal of Systems Architecture - Embedded Systems Design 60(4), 372–392 (2014)

[4] Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Wegener, J.: The impact of input domain reduction on search-based test data generation. In: Proc. of the the 6th Joint Meeting of the European Software Engineering Conf. and the ACM Symp. on The Foundations of Software Engineering. pp. 155–64. ACM, New York, USA (2007)

[5] Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Software Engineeering Journal 11(5), 299–306 (1996)

[6] McMinn, P., Shahbaz, M., Stevenson, M.: In: Search-Based Test Input Generation for String Data Types Using the Results of Web Queries. pp. 141–50. IEEE, N (2012)

[7] McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14, 105–156 (2004)

[8] Mendes, C.C.T.: Navigation of Mobile Robots Using Stereo Vision (in Portuguese). Master's thesis, ICMC/USP, São Carlos/SP - Brasil (2012)

[9] Mendes, C.C.T., Wolf, D.F.: In: Real Time Autonomous Navigation and Obstacle Avoidance Using a Semi-global Stereo Method. pp. 235–36. ACM, New York, USA (2013)

[10] Neves, V.d.O., Delamaro, M.E., Masiero, P.C.: Structural testing of autonomous vehicles. In: Proc. of the 21th Int. Conf. on Software Engineering and Knowledge Engineering. pp. 200–05. SEKE 13, Knowledge Systems Institute, Boston, USA (2013)

[11] Neves, V.d.O., Delamaro, M.E., Masiero, P.C.: An environment to support structural testing of autonomous vehicles. In: Proc. of the the Brazilian Symposium on Computing Systems Engineering. pp. 1–6. SBC, Porto Alegre, BR (2014)

[12] Neves, V.d.O., Delamaro, M.E., Masiero, P.C.: Heuristics for the selection of the initial population of search based data generation algorithms for autonomous vehicles controller software. In: Proc. of the 8th Brazilian Workshop on Systematic and Automated Software Testing. pp. 41–50(in Portuguese). SBC, Porto Alegre, BR (2014)

[13] Nguyen, C.D., Miles, S., Perini, A., Tonella, P., Harman, M., Luck, M.: Evolutionary testing of autonomous software agents. Autonomous Agents and Multi-Agent Systems 25(2), 260–283 (2012)

[14] Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithm. Journal of Software Testing, Verification and Reliability 9(8), 263 – 82 (1999)

[15] Urmson, C., Anhalt, J., Bagnell, D., et al.: Autonomous driving in urban environments: Boss and the urban challenge. Journal of Field Robotics 25, 425–466 (August 2008)