Universidade de Brasília
Decanato de Ensino de Graduação
Diretoria de Acompanhamento e Integração Acadêmica

ÁREA DO CONHECIMENTO: (X) EXATAS ( ) HUMANAS ( ) VIDA

Título do Projeto

Universidade de Brasília
Decanato de Ensino de Graduação
Diretoria de Acompanhamento e Integração Acadêmica

Inteligencia Artificial Aplicada ao Desenvolvimento de Controladores de Veículos Autônomos

Orientador:

Guilherme Novaes Ramos

Unidade Acadêmica/Departamento:

Instituto de Ciências Exatas/Ciência da Computação

Relatório Final

JTCic/UnB 2013

Título do Plano de Trabalho

Desenvolvimento de Controlador PID para Veículos Autônimos

Aluno

Guido Dutra de Oliveira

Matrícula

13/0009733

Universidade de Brasília
Decanato de Ensino de Graduação
Diretoria de Acompanhamento e Integração Acadêmica

Inteligencia Artificial Aplicada ao Desenvolvimento de Controladores de Veículos Autônomos

1. Introdução

O advento da computação tornou possível a automação de diversas atividades antes executadas apenas por humanos ou sob supervisão humana. Com os estudos no campo da inteligência artificial, o escopo de atividades passíveis de automação é cada vez mais abrangente, e a eficiência das automações cada vez maior.

Dentre as aplicações de inteligência artificial mais importantes está a direção de automóveis. Ainda que hoje carros dirigidos automaticamente sejam em sua maioria protótipos, muitos carros comerciais possuem sistemas que realizam partes da direção automaticamente (freio automático, marcha automática, estacionamento automático, etc). Contudo, há exemplos de veículos comerciais completamente automáticos, como o Navia, um automóvel feito para operar em áreas de pedestres desenvolvido pela Induct[1]. A tendência é que o desenvolvimento de automóveis automatizados aumente significativamente nos próximos anos.

Apesar dos avanços recentes, construir um controlador ainda é uma tarefa bastante complexa, levando em conta a quantidade de fatores que devem ser considerados para cada decisão a ser tomada pelo controlador (velocidade do carro, marcha, estado da pista, presença ou não de outros pilotos nas proximidades, ângulo entre o carro e a pista, etc).

Para facilitar o processo de criação de um piloto, é necessária uma maneira de entender como cada fator influencia no comportamento do piloto, e de estudar como mudanças nestes fatores alteram seu comportamento. O intuito do projeto é justamente criar uma maneira de lidar com o problema descrito acima: a proposta é desenvolver uma interface genérica para evolução de controladores.

O módulo foi projetado para controladores no ambiente do TORCS (The Open Racing Car Simulator)[2], um simulador de corrida que se destaca como plataforma de pesquisa e treinamento de I.A por ser acurado (em termos da física da corrida), portável e ter código aberto. De maneira geral, o algoritmo deve atualizar um determinado estado do piloto ao qual foi designado toda vez que um determinado evento ocorre na simulação (quando uma volta é completada, por exemplo).

O algoritmo foi desenvolvido como uma classe abstrata em C++ (mesma linguagem na qual foi criado o TORCS), e foram implementadas três subclasses, cada uma com uma abordagem diferente.

Antes do começo do projeto propriamente dito, foram realizadas atividades com o intuito de familiarização com o método científico e com a escrita de artigos científicos. Será feita uma breve descrição destas.

Foram estudados quatro artigos, dos quais fez-se o fichamento e a apresentação em slide show (exceto para o primeiro, do qual fez-se apenas o fichamento). O segundo artigo foi de livre escolha, enquanto os outros foram determinados pelo orientador.

Os artigos abordados foram, em ordem cronológica:

- “The 2009 Simulated Car Racing Championship” - Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A. Pelta, Martin V. Butz, Thies D. Lönneker, Luigi Cardamone, Diego Perez, Yago Sáez, Mike Preuss, and Jan Quadflieg
- “Fuzzy Sets” - L. A. Zadeh
- “Quicksort” - C. A. R. Hoare
- “A* (a-star)” - Peter E. Hart, Nils J. Nilsson, Berthram Raphael

O fichamento dos artigos foi feito no LaTeX, um sistema de configuração e organização de documentos muito utilizado para a publicação de artigos científicos, e as apresentações em slide no Beamer, classe do LaTeX projetada para a criação de slides. A familiarização com essas ferramentas por si só pode ser considerada parte importante do programa. Os fichamentos foram avaliados pelo orientador e discutidos, assim como os slides, que foram apresentados pelos alunos na sala de reuniões.

Segue a descrição do projeto no próximo tópico.

2. Revisão Teórica

O TORCS, além de ser um excelente simulador de corrida, é um programa de código aberto, o que o torna ideal para o desenvolvimento de controladores, já que dessa forma tem-se maior entendimento a respeito do funcionamento do simulador e da maneira com que este interage com o piloto, e portanto é possível fazer implementações mais concisas e eficientes, além da possibilidade de fazer alterações no código do próprio simulador, caso seja necessário. A qualidade do TORCS é confirmada pela existência de uma corrida simulada anualmente cujos pilotos são inteligências artificiais projetadas para o ambiente do TORCS[3]. Nesta corrida, cada equipe inscreve seu piloto – um controlador desenvolvido a partir do “kit” fornecido pelos organizadores da corrida – e os pilotos competem em corridas simuladas no TORCS.

O programa desenvolvido foi testado utilizando-se o *PIDDriver*, piloto baseado no conceito matemático *PID (proportional-integral-derivative) controller*, um mecanismo de controle cuja função é o ajuste de um determinado parâmetro a partir do erro deste em relação a um valor esperado. Uma descrição mais completa do PID será dada em seguida. O *PIDDriver* foi criado pelo orientador a partir do piloto disponibilizado por um dos organizadores dessa corrida, o *simple driver*, de maneira a garantir a usabilidade do programa para outros pilotos no TORCS.

O TORCS foi escrito em C++, uma linguagem baseada em C, porém capaz de lidar com programação orientada a objetos, ou seja, estão definidos os conceitos necessários para este paradigma (classes, objetos, herança, métodos, etc).

Uma *classe* em programação orientada a objetos é um molde para a criação de *objetos*, possuindo valores iniciais para os *estados* (variáveis) e implementações de *comportamento* (funções/métodos). Ou seja, uma classe é um molde do qual pode-se criar uma instância, a qual é chamada *objeto*, e representa uma entidade que possui estados e comportamentos específicos. A abordagem escolhida foi criar uma classe que represente o módulo de atualização. Essa classe deve poder ser instanciada por um piloto qualquer e deve possuir um método que execute alguma mudança neste piloto durante o

decorrer da corrida dado que algum evento ocorreu (o módulo passou por algumas etapas antes da criação da classe propriamente dita. Estas serão vistas na próxima seção).

No caso do PIDDriver, a mudança a ser executada é a atualização das constantes associadas aos três parâmetros do controlador PID. Essas constantes definem o peso associado a cada parâmetro do controlador, o que altera a intensidade da ação corretiva do controlador em relação à variável à qual ele está associado (na versão do piloto utilizada para os testes o controlador estava associado à velocidade do carro, sendo que um valor fixo foi escolhido como velocidade desejada, e o controlador portanto adaptava a velocidade atual do carro de maneira a aproximá-la dessa velocidade fixa).

Pela definição de um controlador PID, temos que: P (*proportional)* representa o erro atual entre o valor verdadeiro e o valor esperado da variável, I (*integral)* representa o somatório de todos os erros medidos, e portanto considera a *duração* do erro, e D (*derivative*) representa a taxa de variação do erro, prevendo o comportamento futuro da variável.

As constantes de peso associadas a esses parâmetros são geralmente chamadas de *kp, ki e kd,* respectivamente, e são estes valores que devem ser atualizados pelo módulo a ser projetado. Vamos ver detalhadamente como este módulo foi implementado na próxima seção.

3. Desenvolvimento do Módulo de Atualização

Primeiramente foi estudada a maneira pela qual um piloto qualquer se comunica com o simulador (TORCS). Esta interação se dá por intermédio de um método chamado “drive” que deve ser implementado pelo piloto e que é chamado pelo simulador a cada 0.02 segundos. Esse método recebe do simulador a situação atual do carro por meio dos seus sensores e retorna a ação que será tomada pelo piloto (virar x graus à esquerda, mudar de marcha, acelerar. Etc).

A primeira versão do módulo de atualização foi criada diretamente no código do *PIDDriver*, como um método interno do piloto que é chamado pelo método drive e atualiza um dos parâmetros (*kp, ki ou kd*), que estão armazenados em um objeto do tipo *PIDController* contido no piloto. A atualização dos parâmetros foi feita de maneira que estes variassem em um determinado intervalo (foi escolhido o intervalo de 0.001 a 1), e a cada atualização um dos parâmetros fosse incrementado (o incremento é de 0.001). Quando o primeiro parâmetro chega ao seu valor máximo ele retorna para o menor valor e o segundo parâmetro é incrementado uma vez. Igualmente para o terceiro parâmetro.

Nessa implementação a atualização ocorre a cada chamada do método drive, de maneira que não há muita utilidade prática, dado que uma única interação entre o piloto e o simulador produz uma alteração tão mínima que é irrelevante para a corrida como um todo. Após a verificação da funcionalidade adequada da função foi adicionada uma condição de atualização. Esta foi a mais básica possível: a cada x chamadas da função drive, um dos parâmetros é atualizado. Nesse caso x é passado como parâmetro para o método de atualização, que é chamado pelo método drive e que possui um contador que é incrementado a cada atualização. Caso o contador atinja o valor de x, a atualização é executada e o contador volta para 0.

Além dessa condição, foi implementada ainda uma outra, cujo intuito é realizar uma atualização a cada volta do piloto. Neste caso, a função recebe como parâmetro a referência ao objeto do tipo *CarState,* que possui todas as informações relevantes sobre o estado atual do carro. Um dos métodos da classe *CarState* é o método *GetCurrentLapTime,* que retorna o tempo que decorreu desde o início da volta atual. Este método é usado pela função de atualização para descobrir se o piloto completou uma volta. Em caso positivo, a atualização é implementada.

Essas duas condições podem possuir certa utilidade em alguns casos (pode-se medir o desempenho do carro em trechos específicos da pista com diferentes configurações, ou medir o desempenho de cada volta, por exemplo), porém são apenas exemplos simples que podem e *devem* ser modificados pelo usuário de acordo com suas necessidades, de maneira que o módulo possa ser utilizado em diferentes situações para resolver problemas específicos de cada usuário.

O próximo passo na implementação foi tornar o módulo de atualização independente do módulo do piloto. Para isso, o código alterado do *PIDDriver* foi utilizado como base para a criação de uma nova classe, denominada *Updater,* que, sendo instanciada por um ploto qualquer, deve implementar a atualização, sem que seja necessário alterar significativamente o código do piloto.

A classe *Updater* foi concebida como uma classe abstrata que possui apenas dois métodos: *update* e *directUpdate.* A ideia é que o método *update* implemente a verificação da condição de atualização, ou seja, se é ou não necessário promover a atualização, e, caso seja, chame o método *directUpdate,* que implementa essa atualização. Caso o usuário deseje promover a atualização diretamente, sem testar nenhuma condição, pode chamar diretamente o método *directUpdate.* Para que um piloto qualquer utilize o módulo, basta incluir no header alguma subclasse de *Updater* e instanciá-la no construtor do piloto. Após isso, os métodos podem ser utilizados pela função “drive” do piloto.

Sendo *Updater* um método virtual, é necessária a criação de subclasses que implementem os métodos virtuais *update* e *directUpdate.* Foram criadas três subclasses, todas projetadas para a utilização no *PIDDriver,* mas cada uma com implementação ligeiramente diferente. As três subclasses possuem uma referência para um objeto do tipo *PIDController*, que é inicializado no construtor com a referência ao objeto dessa classe contido no piloto. Segue uma descrição de cada uma:

- *LapUpdater*: a cada volta realizada pelo piloto, um dos três parâmetros (*kp, ki e kd)* é atualizado. As funções *update* e *directUpdate* recebem como parâmetro um objeto do tipo *CarState*.
- *CountUpdater*: a cada *x* chamadas da função *update* um dos três parâmetros (*kp, ki e kd)* é atualizado. Esta subclasse possui uma variável maxCount, que é inicializada no construtor com o valor de *x.* As funções *update* e *directUpdate não* recebem parâmetros.
- *SingleUpdater*: a função de verificação é igual à de *LapUpdater*, porém apenas um dos parâmetros é alterado, Independentemente dos valores atuais. O parâmetro a ser alterado deve ser especificado como parâmetro para as funções *update* e *directUpdate*, na forma de um char: *“p”* para *kp*, *“i”* para *ki*, e *“d”* para *kd.* Caso outro char seja recebido, as funções escrevem uma mensagem de erro e retornam sem fazer alterações.

A função *directUpdate*, em todas as implementações, escreve no console a mensagem “Updated!” e os valores atualizados dos três parâmetros, quando a alteração é feita de maneira correta.

4. Resultados

Segue uma tabela dos resultados finais de corridas utilizando o piloto *PIDDriver*. Em cada campo o piloto utiliza uma subclasse diferente da classe *Updater*. As corridas foram feitas no circuito “E-tracks 5” em cinco voltas. O “E-tracks 5” foi escolhido pois é um campo oval e de pequeno tamanho, o que elimina a possibilidade do piloto sair da pista, o que poderia atrapalhar os testes, uma vez que o piloto não possui um módulo de retornar à pista, podendo ficar preso por um tempo considerável. Foram determinadas cinco voltas por corrida arbitrariamente, sendo que o número de voltas (desde que maior que um) não influencia muito os testes. A velocidade desejada do carro, utilizada na função do controlador PID, foi definida como 130 para a execução dos testes, novamente para impedir que o piloto saia da pista e fique preso, dado que uma velocidade muito superior a essa poderia levar a esse comportamento. Para a classe *CountUpdater*, foi utilizada a quantidade de 200 ticks entre cada atualização, e para *SingleUpdater* foi escolhido que se atualizasse sempre o valor de *kp*.

piloto (subclasse de updater)	total de atualizações	tempo total da corrida	tempo da melhor volta	velocidade máxima atingida
sem alterações	0	03:46:30	44:94	184
LapUpdater	4	03:46:19	45:10	187
CountUpdater	53	04:46:18	45:05	187
SingleUpdater	5*	04:46:19	45:09	187

Em todos os casos, *kp, ki e kd* foram inicializados com os valores 0.001, 0.05 e 0.005, respectivamente. Na versão do *LapUpdater*, *kd* foi incrementado a cada atualização, terminando em 0.009. Na versão do *CountUpdater*, *kd* é atualizado da mesma forma até 0.059. Na versão do *SingleUpdater*, *kp* é atualizado, começando em 0.001 e terminando em 0.006. Foram feitos outros testes, em que os valores iniciais das três variáveis foram alterados, para checar se a atualização de todas as variáveis ocorre de maneira correta em todas as implementações.

Nota-se que a versão do *SingleUpdater* promove uma atualização a mais do que a versão do *LapUpdater*, o que é inesperado, já que a condição de atualização é a mesma para as duas classes. Porém, essa atualização a mais na primeira classe é realizada justamente no momento em que a corrida termina, quando o piloto passa pela linha de chegada após a quinta volta, e uma nova volta começaria (portanto essa atualização seria correta) caso não fosse o fim da corrida. O por quê dessa atualização ocorrer no *SingleUpdater* mas não no *LapUpdater* não pôde ser encontrado, porém trata-se apenas de uma questão de prioridade entre terminar a corrida e promover a atualização, o que de forma alguma altera o resultado final do teste.

Em outra análise, é importante notar que as diferenças entre os resultados das implementações diferentes são sutis, pois os testes foram realizados em uma pista oval, na qual o piloto tende a manter uma velocidade constante (já que não há necessidade de frear), o que acaba estabilizando bastante o controlador PID, impedindo as mudanças de parâmetros de provocarem mudanças muito drásticas de comportamento. Mais do que avaliar o quão impactantes foram as mudanças, deve-se apenas perceber que os resultados foram de fato diferentes para os tipos diferentes de implementação, ou seja, a alteração das variáveis forçosamente influenciou no comportamento do piloto.

Se essa influência foi benéfica ou maléfica, sutil ou drástica é uma questão fora do escopo do projeto. O importante é notar que as alterações feitas influenciaram o resultado, ou seja, o módulo funciona e pode ser implementado em outros pilotos sem grandes dificuldades. Um exemplo da utilização do módulo seria fazer diversos testes e analisar para quais valores das três variáveis o desempenho do piloto é mais satisfatório, e a partir desse resultado otimizar o potencial do piloto.

5. Conclusão

A proposta do projeto foi elaborar uma maneira de amenizar a dificuldade de criar um controlador evolutivo por meio de uma interface genérica que pudesse ser acoplada a um controlador qualquer, e que permitisse promover mudanças no piloto sem alterar muito o código fonte.

No fim do projeto, a solução encontrada foi a criação de uma classe abstrata com dois métodos, um de verificação e um de atualização. Essa abordagem permite a reutilização do código para outros pilotos que não aquele com o qual o módulo foi testado. Os testes sugerem que o módulo funciona adequadamente, fazendo as verificações e atualizações de maneira coerente.

Apesar de extremamente simples, o módulo de atualização implementado pode tornar-se uma ferramenta útil para o desenvolvimento de controladores. Com três linhas de alteração no código do piloto – uma para a inclusão do módulo, uma para a instanciação e uma para chamar o método – é possível adicionar o módulo a este, como foi feito com o *PIDDriver (vide a seção 3)*, o que demonstra a sua facilidade de uso.

A proposta do módulo foi cumprida de maneira satisfatória. Quanto aos problemas gerais relacionados à otimização do piloto (como armazenar os dados atualizados, que condição de atualização utilizar, como aproveitar-se do módulo para mais de uma corrida, etc.), cabe ao usuário planejar uma maneira de contornar estes empecilhos e beneficiar-se de maneira inteligente do módulo.

6. Referências Bibliográficas

[1] http://induct-technology.com/en/products/navia-the-100-electric-automated-transport
[2] http://torcs.sourceforge.net

[3] “The 2009 Simulated Car Racing Championship” - Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A. Pelta, Martin V. Butz, Thies D. Lönneker, Luigi Cardamone, Diego Perez, Yago Sáez, Mike Preuss, and Jan Quadflieg