Autonomous Agents:
A Hybrid Dynamical System for Vehicle Navigation
Comp 352

Micah Wylde
Jeffrey Ruberg

May 3, 2011

1 Introduction

Self-driving cars hold much promise for saving fuel, time and lives. Human-driven vehicles are responsible for millions of traffic accidents and over 30,000 deaths annually in the US alone. Humans can also make poor decisions in regards to routes, leading to traffic jams and wasted fuel. Safe, effective autonomous cars could largely solve these issues. AI researchers have been interested in the potential for vehicular navigation since the 1960s. In order to spur development in this area, the Defense Advanced Research Projects Agency (DARPA) organized the Grand Challenge in 2004, which tasked contestants to build cars which could autonomously navigate a 142 mile-long course in the Mojave desert. Though none of the vehicles could finish the course, a second competition the next year was more successful. Four teams finished in the allotted time, traversing a treacherous 132 mile desert course with no human guidance. Building on this achievement, in 2007 DARPA organized the Urban Challenge which took place in a simulated suburban environment. To win, cars had to navigate a maze of streets while accounting for other traffic and following California traffic laws at all times. Six teams finished the 61 mile course with the winner, "Boss" from CMU, taking a little more than four hours [**?**].

There are many challenges involved in autonomous driving and robot navigation in general. In the real world one must deal with unreliable and imperfect sensor data, imprecise localization technologies and other perception issues. Even in simulation the challenges of successful navigation are immense. Cars must follow traffic laws, get to their destination quickly and efficiently and act safely at all times, even in unexpected situations. As far as the actual navigation, there are two general approaches: deliberative and reactive. As an example of the former, A^* is an efficient and optimal graph search algorithm which is very effective at finding the best route between two points but cannot handle the dynamism of the real world. Dynamical systems-based navigation is a reactive strategy that uses force fields to guide agents away from obstacles and towards their target. However, it is purely local and has trouble finding optimal paths to distant goals. In this paper we present simulated agents which use each technique as well as a hybrid agent which makes use of A^* for global navigation and dynamical systems for local navigation.

2 Autonomous Driving

2.1 Simulation

3 Methods

To create an autonomous agent that navigates while simulating a car's behavior and traffic laws, we took three general approaches: deliberative planning through A^* search, reactive navigation through a dynamical system, and a hybrid of deliberative planning and reactive motion. The system architecture consists of a server and a separate client for each agent.

3.1 Code Architecture

The project is written in Ruby, under JRuby to utilize Java2D for the graphical display. Specifically, all server code runs solely under JRuby, but client code can run under any flavor of Ruby. Agents are represented both on the client and server end; server agents perform motion- and display-related calculations, and client agents contain all the navigation inference and decision-making and ultimately send decisions (restricted to behavior variables) back to the server againt.

A brief description of the roles of the various source files will follow.

app.rb This file is the point of entry for the program. This same entry point is used to, based on various command-line options, start a server, start a new agent, or run tests.

client_agent.rb This file contains the `ClientAgent` class, the super class which every client agent inherits. Client agents receive messages from remote server agents, process the message (based on their form of navigation), and then send back a response in their behavior variable space.

constants.rb This file contains various general constants that may be used in several locations or files, or that may be particularly useful to tweak with.

display.rb This file contains all of the display code used to generate our rendering of the world.

map.rb This file contains the classes, specifically `Map`, which encode information provided from real map data.

pqueue.rb A priority queue implementation useful for A^* search.

remote_agent.rb This file contains the `RemoteServerAgent` class, which is a subclass of `ServerAgent`. Essentially, a remote server agent is a server agent which is tied to a specific client agent and communicates with that client agent.

server_agnet.rb This file contains the `ServerAgent` class, which contains all the base representation and calculations needed for an agent (for example, the server agent computes various points needed to display the agent graphically).

server.rb This file contains the socket server to which new agents connect.

socket.rb MICAH

util.rb This file contains a collection of geometry classes (`Point`, `Vector`, etc.) which are useful in the display and other various places (most notably in dynamical navigation calculations).

agents/astar.rb This file contains a client agent that deliberatively plans paths using A^* search.

agents/dynamical.rb This file contains a client agent that navigates through a purely dynamical system-based approach.

agents/hybrid.rb This file contains a client agent that navigates through a combination of deliberative planning and dynamical systems.

agents/simple.rb This file contains a very primitive client agent (that can hardly be called an agent) which allows us to easily test the motion calculations performed by server agents.

3.2 Simulation

To provide a realistic environment for navigation tests, we used real-world map data from OpenStreetMap.org, which provides XML-formatted maps for the entire world. The data format, called OSM, provides a graph representation of a road system; streets are represented by placing nodes wherever the road turns or intersects other roads, with edges connecting each node. To provide testing data, we used the website to export maps of Hayward, CA and Santa Cruz, CA. We wrote a program, `osm_convert`, which converts an OSM XML file to a YAML[1] format. In `map.rb` we then read this YAML data and construct an internal representation of the graph, converting points specified by latitude and longitude to a scale of meters.

In order to make an internal representation of the map suitable for a driving simulation, we generate a *road* for each graph edge by creating *walls* defined as parallel lines a constant `ROAD_WIDTH` away from the center line, which is the line connecting the node and the node the edge connects it to. This parallel line approach creates overlapping walls at every acute angle and gaps at every obtuse angle, so we process the walls to clip them where they overlap and extend them where they leave gaps.

To run the display, we start a server by running the script `bin/driving` without any command line arguments (or with arguments to specify window geometry or map file). In the display, we have implemented mouse dragging, zooming, pausing, agent centering (on by default, toggled with space bar), and agent placement/manipulation.

To create an agent, we create an agent process (while a server process is running) by running the script `bin/driving` with command line argument `-c` or `--agent` specifying the name of the class of client agent to create (`AStarAgent`, `DynamicalAgent`, `HybridAgent`, or `SimpleAgent`). The agent will then be shown on the display, and we can rotate among all current agents with the arrow keys if agent following is enabled in the display. When a client agent is created, a paired remote server agent will be created; the remote server agent will constantly send its current state information to the client agent, which will respond with its decision.

We intended to model the agent's motion using basic, low-speed car physics, which conclude that a car with a set wheel position (set to anything other than straight) will trace out a circular path. The path is determined by δ (the angular displacement of the wheel, with $\delta > 0$ indicating a right turn), the length of the car, and the speed the car is traveling. We have, for practical concerns which will be discussed shortly, resorted to a much more primitive behavior variable space, where agents choose their heading angle directly.

3.3 Deliberative Planning

3.4 Reactive Navigation

- explain why choosing phi instead of delta is actually okay here

[1]YAML (a recursive acronym for YAML Ain't Markup Language) is a language-independent data serialization standard which allows easy conversion of data structures to and from a string representation.

3.5 Hybrid Approach

4 Results

5 Conclusion

5.1 Unreached goals

- cached map rendering - realistic choice variables