Kalman Filter Lab

CS 470 Lab 3

Duane Johnson, Michael Ries and Jon Brandenburg

June 5, 2009

Abstract

This lab is focused on learning the basics of implementing a Kalman filter. The participants had to rely on 'noisy' sensors which added normally distributed noise to each dimension independently. In order to evaluate the perfomance of the Kalman filter the participants had to implement an agent which would try to shoot another tank based on the filtered value of where it believed it was and where it believed the other tank was. Finally the participants implemented a set of 'clay pigeons' with different, but well-defined behaviors to help them assess how the Kalman filter was behaving.

Contents

Chapter 1

Lab Context

1.1 Summary

The authors each developed and ran the software below on laptops. The laptops were unix-based systems; however, these results should be easily duplicated on a Windows machine.

1.2 Hardware

- Macbook Pro (32 bit) / 2.4 GHz Intel Core 2 Duo / 2 GB RAM
- Lenovo T400 (32 bit) / 2.2 GHz Intel Core 2 Duo / 2 GB RAM
- HP Pavilion (64 bit) / 2.5 GHz Intel Core 2 Duo / 4 GB RAM

1.3 Software

- Mac OS X 10.5
- Ubuntu 9.04
- bzFlag 2.0.7 with patched bzrobots client[1]
- Ruby Interpreter, version 1.8.7 (or 1.9.1)

[1]Supplied by CS470 in *Install Instructions*

- EventMachine Library[2] for Ruby
- PDF-Writer Library[3] for Ruby
- Rake Library[4] for Ruby Unit Tests
- NArray and NMatrix [5] for Ruby

[2] http://rubyeventmachine.com/
[3] http://ruby-pdf.rubyforge.org/pdf-writer/
[4] http://rake.rubyforge.org/
[5] http://narray.rubyforge.org/

Chapter 2

Experiences and Time Spent

2.1 Duane's Summary

The majority of my time this week was spent optimizing our A^* algorithm. Our original implementation was entirely in Ruby, but it turned out to be too slow for discretization below 40 meters or so. The new code is a Ruby extension written in C, and performs much better (1/2 second compared with 45 seconds on a 1000×1000 grid).

I also refactored our agent code to use state transitions for this lab. Rather than using several different Agent classes for each task (e.g. a 'dummy' agent vs a 'smart' agent), we unified our code into a single agent that could make decisions regarding states. Once this refactoring was in place, Jon Brandenburg specialized our agents for sniper and decoy capabilities.

I spent about 18 hours on this project, divided as follows:

- 8 hours implementing a fast A^* algorithm in C
- 3 hours refactoring our agents to use state information
- 3 hours building a new PDF output of our map with tank and flag positions
- 2 hours debugging and passing off with another team in the CS Sports lab.
- 2 hours writing this report and formatting it in LaTeX

2.2 Michael's Summary

This lab was especially fun because once again we got to see our agents take action in the BZFlag world. We now had the ability to see what our agents were 'thinking' by using Duane's code to generate PDF maps of how the agents saw the world. We also had the ability to watch them act by joining the BZFlag world as an observer. This made it much more insteresting to work on because we could see immediate results.

Duane was the first to make serious progress with his A^* module written in C. This gave Jon and me a good place to start implementing our parts. I focused on implementing the path-following code and using potential fields. The original implementation would always place an attractive potential field on the next node in the solution path, but that made our agents very slow and tentative because the individual path segments were very small. This gave rise to the idea of placing the attractive field further ahead. This eventually turned into the algorithm that places the Potential Field as far ahead as possible before encountering a turn.

In the end I spent a total of about 20 hours split across the following tasks:

- 4 hours of implementing the initial path-following algorithm to place potential fields and use the new data structure that represents the world.
- 2 hours meeting with team before class to talk about solving problems related to discretizing the world and speeding up the process of marking blocked nodes
- 2 hours of changing our algorithm to evaluate which nodes are blocked
- 4 hours optimizing the path-following algorithm and tuning the strength of our attractive fields
- 4 hours meeting with team to integrate final code and prepare for passof
- 2 hours passing off with other team
- 2 hours compiling report materials

2.3 Jon's Summary

I worked on the sniper/decoy code. The difficult part of this lab was transitioning to another team's code base. I got a late start since it took longer than anticipated to finish up the previous lab which left me with little time to work on completing this lab.

In terms of time I spent somewhere between 15-20 hours on the project with about ten of those hours being spent writing/tweaking code, the rest being spent writing the report, thinking about it, demonstrating/observing, emails and team meetings.

Chapter 3

Details

3.1 Path Following and Potential Fields

While developing our search algorithms in the last lab we realized that doing full A^* searching was not going to be possible in ruby if our discretization was too granular. To alleviate this problem Duane Johnson developed a small library written in C which acted as a module to the rest of our code. We also simplified the representation of the discretized map for the algorithm to search through. With these changes we were able to search through maps of 80x80 in approximately 1/10 of a second. This was fast enough and granular enough for the purpose of finding our way through a maze and deploying a decoy-sniper tactic.

When it came time to follow the solution path returned by Duane's A^* module the immediate thought was to use potential fields along the length of the path. The potential fields code developed in the first lab already had a PD controller which would suggest velocity and angular velocity based on the current position and angle of the tank. As the lab progressed we optimized this algorithm to actually look as far ahead as the line segments are parallel so that when driving in a straight line we don't spend much time re-evaluating the path. We also periodically re-evaluate the search solution which will be important in the future when we are trying to avoid enemy tanks and our world becomes more dynamic.

In addition to the path following optimizations we also 'tuned' the relative strength of the fields and their radii so that as we approach the next Potential Field the system will move it to the next location before our arrival. This

helped to avoid uneccesary 'braking' and turning.

3.2 Schizophrenic Agents

Multiagent searching often times becomes a process of managing, coordinating and controlling schizophrenic agents. Each agent is required to play multiple roles, and each of those roles is broken into a series of discrete steps or states. Each role has a different objective and our task has been to design and coordinate each agent so that they behave in a cohesive manner. We designed our agents such that they used a series of state machines and transitioned from one state to another as each objective was met within the current role.

3.2.1 Agent Design

We designed the following three state machines: search, decoy, sniper. The 'search' and 'decoy' state machines correspond with the roles defined by the lab specification while the 'search' machine recycles the common task of "going to location x, y". The states and their corresponding descriptions can be found in tables 3.1, 3.2, and 3.3. We tried to follow the KISS principle in designing these state machines. Initially there had been some thought to write higher level intelligence into our agents, but the realization came quickly that, in order to leverage existing potential field code, it would be better to write weak AI agents for now. The resulting agents are flexible enough for our current goals, and can be augmented for the increased challenges we anticipate in upcoming labs.

The decoy and sniper state machines both rely on using attractive potential fields to move the tank into position. This was the easiest choice for moving the tanks since the logic to drive a tank based on feedback from potential fields was a solved problem. This choice proved effective in the end but met some initial problems as some latent bugs had to be rooted out to correct the misbehaving agents. Once this was accomplished the tanks moved in a fairly predictable and smooth fashion.

Table 3.1: State Machine - Smart

State	Description
smart	Sets the goal for the tank to the exit point of the maze and moves the tank through the maze
look_for_enemy_flag	Retrieves the enemy flag
return_home	Sets the goal to the home, generates a search path and returns to home

Table 3.2: State Machine - Decoy

State	Description
move_to_start	Sets the destination to the decoy starting position and moves the tank to that point
move_to_goal	Sets the destination to the opposing end of the line and moves the tank to that point

3.3 Agent Behavior

In order to map out the paths to their respective destinations, each agent is initially set to a search state which we labeled "smart". From there, the state transitions bifurcate for each tank—based on the tank ID, it takes on its respective role and continues through the following agent states.

3.3.1 Decoy

The key to meeting the objective of this lab is to ensure that the decoy agent stays closer to the enemy tank as well as keeps from getting shot. With those two objectives in mind we designed the decoy agent to travel back and forth in a straight line between two ends of the field while maintaing a x-distance of roughly 270 meters. Therefore we created two states, one to drive us to the starting point (*move_to_start*) and the second to drive us to the opposite point (*move_to_goal*). If an enemy tank is still alive, our tank would move back to the starting point and repeat the cycle. If all enemy tanks are destroyed, however, then the decoy tank would proceed back to the base by

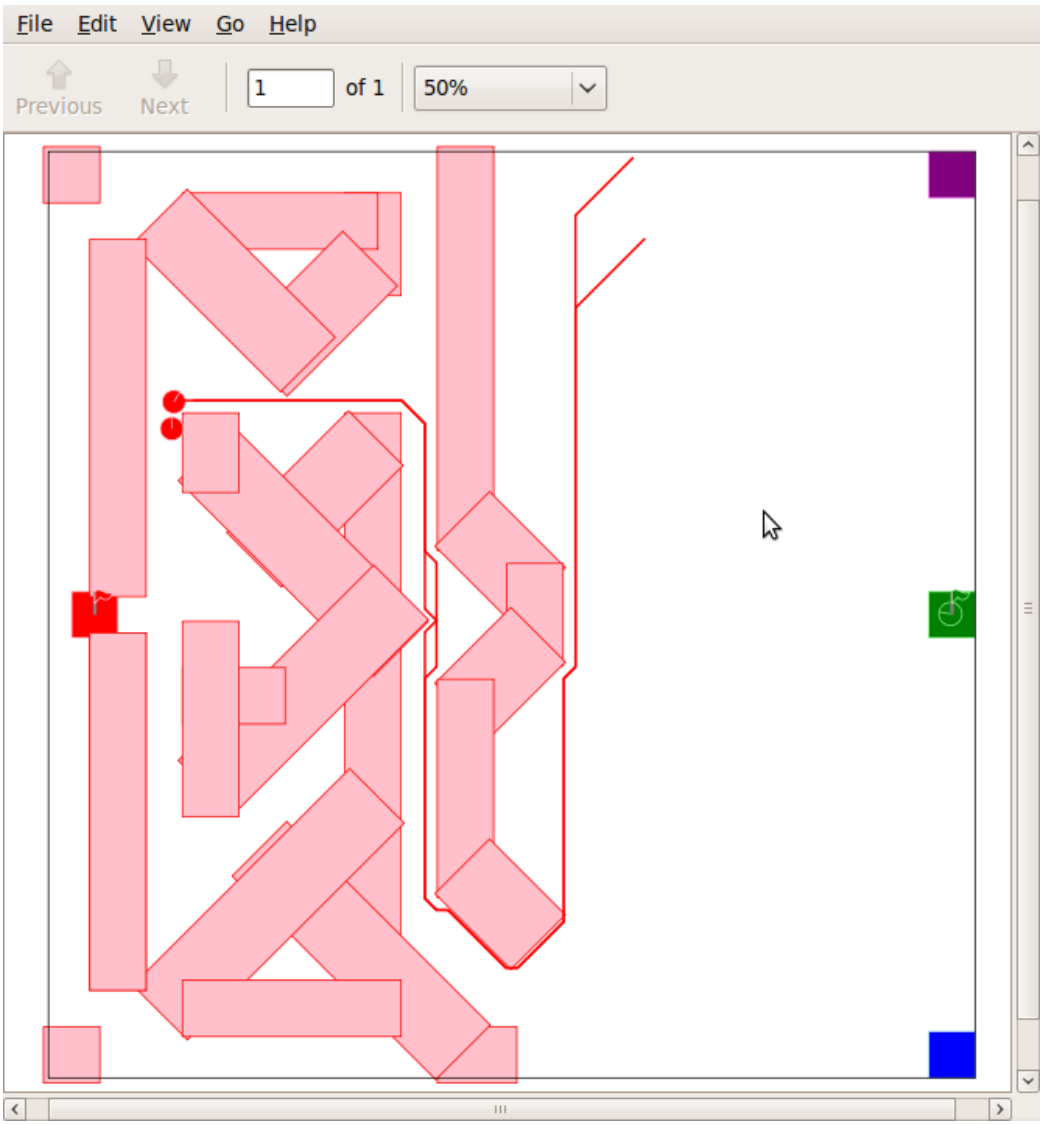

Figure 3.1: Red agents search out a path to their destinations using the *smart* state.

Table 3.3: State Machine - Sniper

State	Description
move_to_start	Sets the destination to the sniper starting position and moves the tank to that point
move_to_attack	Sets the destination to the attack position and moves the tank to that point
attack	Snipes the enemy tank(s).

setting the goal to 'home base' and switching to the *smart* state machine.

3.3.2 Sniper

The sniper agent is designed to move into an initial starting position just outside of the enemy tanks' firing range. The agent polls periodically to see when the distance between the decoy agent and the enemy tank is smaller than between the sniper and the enemy tank. When the sniper detects that the decoy is closer, the sniper transitions from *move_to_start* to *move_to_attack*. At this point the sniper moves approximately 5 meters inside of the enemy tanks firing position. Once the tank has arrived into position it transitions to *attack*. While in attack mode, the tank stops its speed, turns into position and once the proper angle is set, fires a shot, thus destroying the enemy tank.

3.3.3 Looking to the Future

In completing this lab a few shortcuts were made which will need to be addressed in future labs. We relied on the fact that the map orientation (i.e. location of the bases, enemy tank locations, and clear location of the world) were in fixed locations. By making these simplifying assumptions we saved ourselves from writing a bit of code which would have to map out the best path for the decoy, the ideal starting position for the sniper and the ideal sniping position. These simplifications will obviously have to be addressed if our agents are to be successful in more mobile enemy tanks.

There is still some room for correction in the controlling of our tanks via potential fields and we would probably benefit by adding in a PD controller to dampen our movement since there are times when the tank does not slow

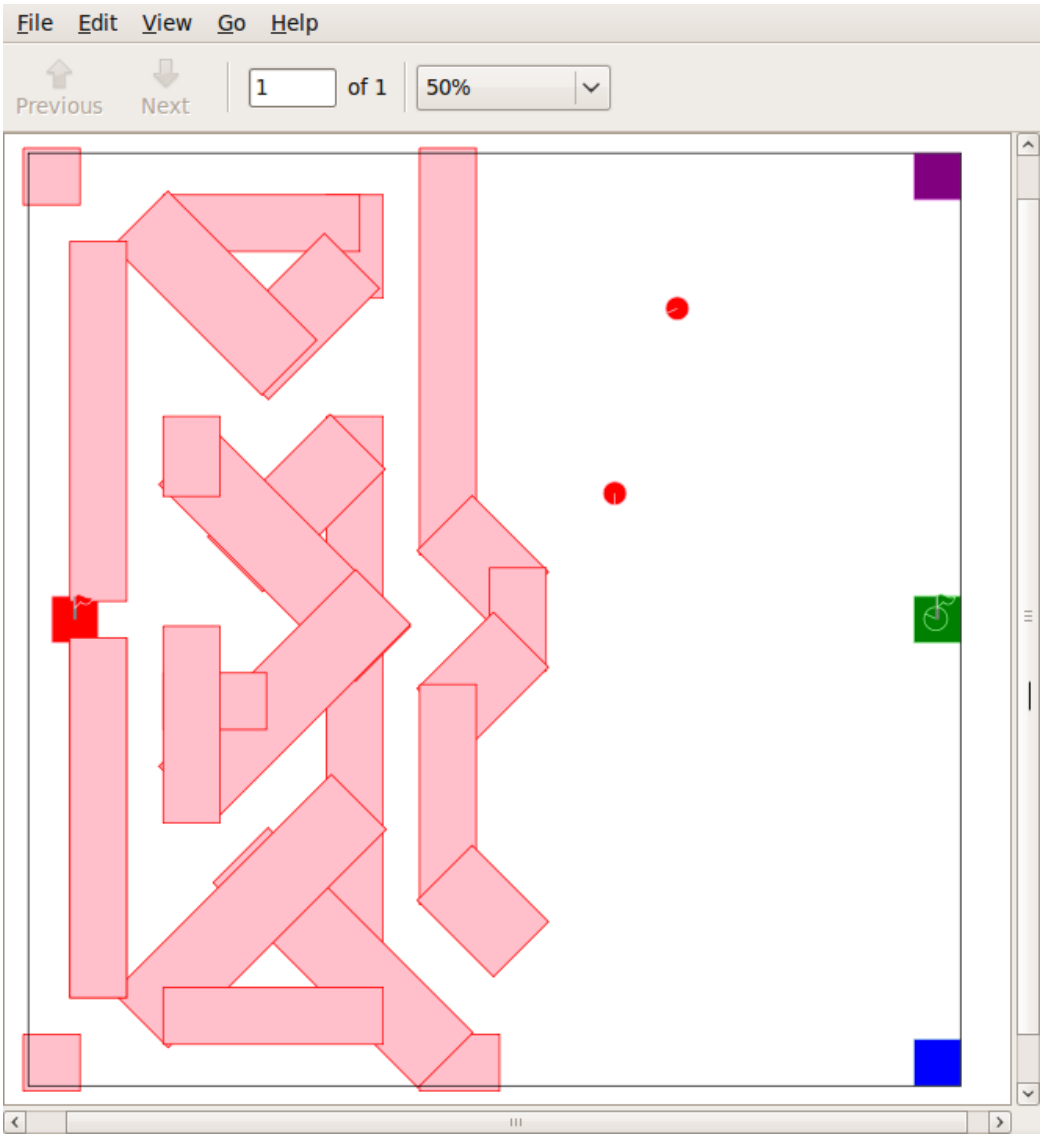

Figure 3.2: The sniper agent moves into place (top) and the decoy agent begins its movements back and forth (bottom).

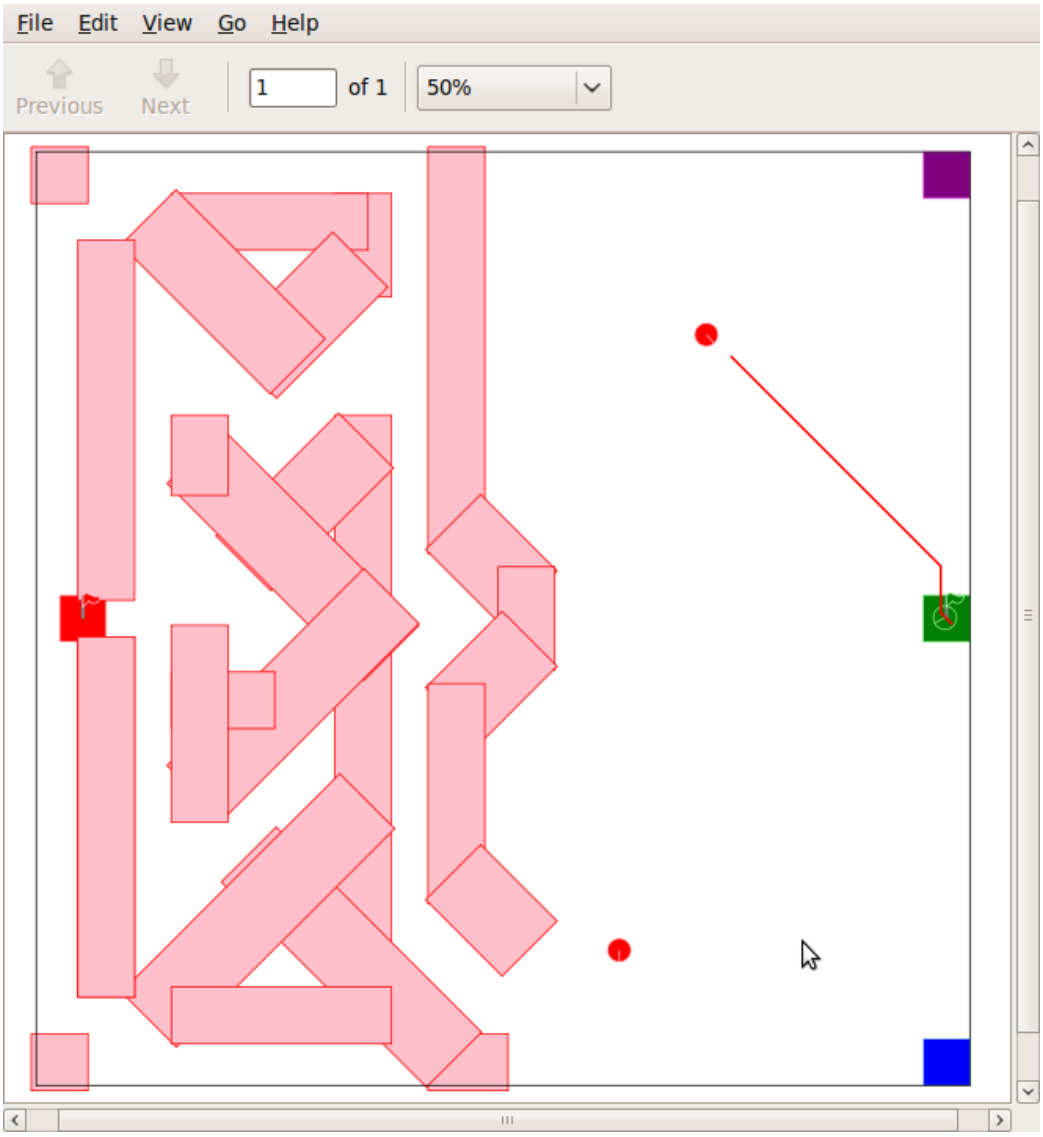

Figure 3.3: After killing the enemy tank, the sniper agent plots a path to the green flag.

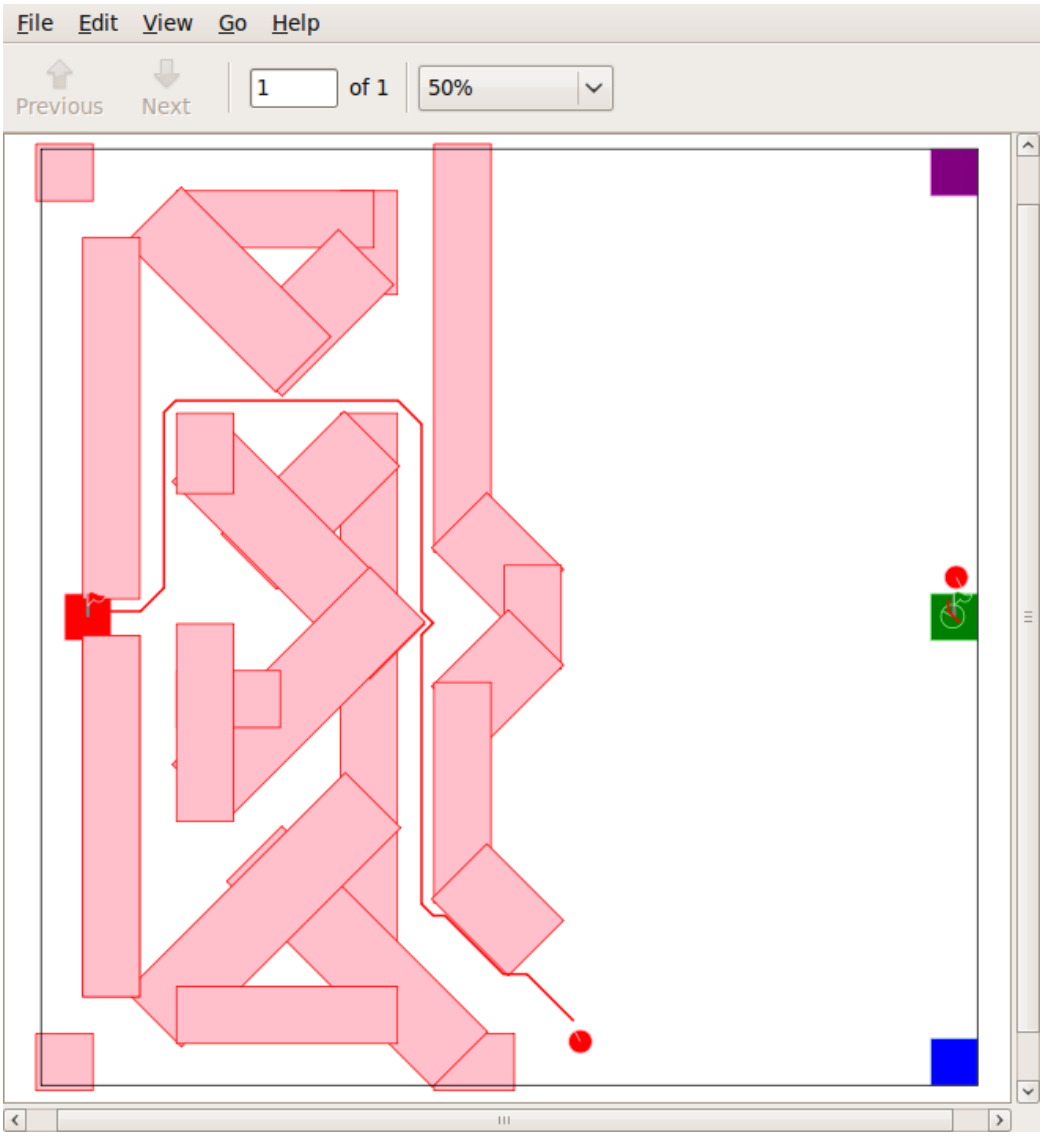

Figure 3.4: The decoy agent returns to home base while the sniper agent picks up the enemy flag.

down or turn in a smooth fashion. Our tanks at times seem to favor turning in a certain direction, which points to a bug in our code. By fixing this I believe most, if not all, of our agents driving problems would be solved.

We made another simplification by not coordinating the efforts between the sniper and the decoy, but instead relying on the timing and initial positioning to keep our sniper safe. However when faced with more tanks, the sniper does not currently move in closer to engage multiple targets if they are not all reachable when it moves into attack mode.

3.4 Tests with Another Group

3.4.1 Our Tests

We ran the tests for our implementation on the *astar.bzw* map from the lab website. Our tests went well with the exception of the return trip to our base after capturing the flag. This is the one odd behavior for which no solution has yet been found.

While demonstrating the code we ran into two difficulties. The first difficulty came when the sniper tank missed picking up the flag and then proceeded to try to return to base. The second came after we fixed the code and reran the agents. After the agent picked up the flag it used the wrong field for a while until the agent realized that it's search. Once the search path was re-evaluated it returned to home base successfully.

While our code demonstratably has some fragile edges, it appears from having observed several other teams code that ours is not alone, however, to its credit it did manage to satisfy the requirements for the lab.

3.4.2 Michael Hansen and Daniel Nelson

We ran our tests with another group from class which consisted of:

- Michael Hansen
- Daniel Nelson

What was interesting about watching the other team pass off was that they had a few problems in their implementation, but they were in no way related to the problems we encountered. The main problem the other team had

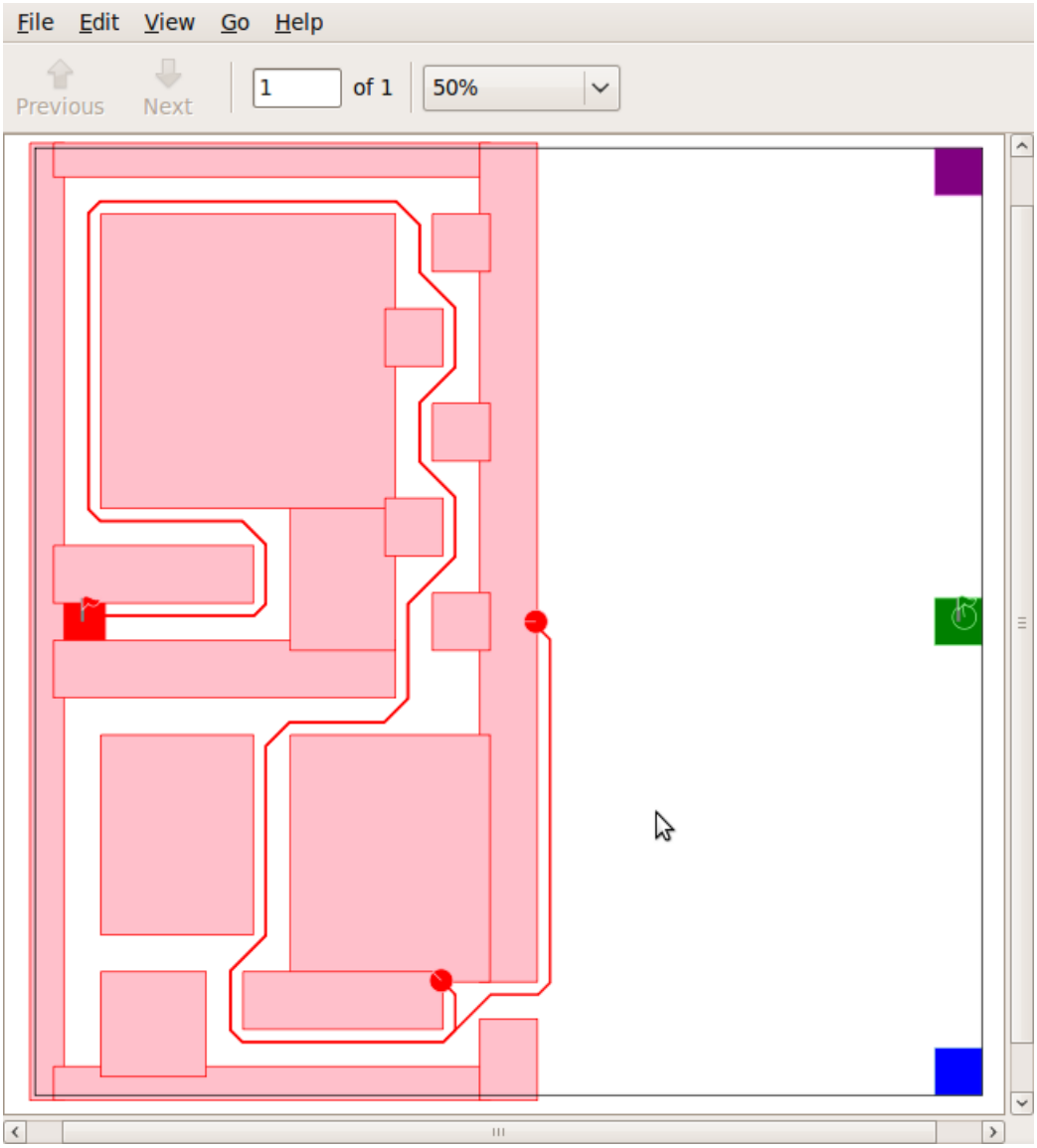

Figure 3.5: Getting stuck at the wall on the way home. Re-evaluated path shown as red line to red base.

was that the decoy would get killed as it went into its decoy mode. They had to change the timing of their agents to avoid this, but in the end they successfully completed the challenge.

3.4.3 Kendall Clement et. al.

In addition to Hansen and Nelson's group, we watched Kendell demonstrate his code. It was quite entertaining because while his team had already passed off his bot was defeated a couple of times before it finally worked. When the bot was able to finally successfully kill the agent, it grabbed the flag and eventually made its way back to the base; however it skirted the edge of the base and as a result the system did not register the capture—presumably because his tank never entered the circle inside of the base.

The other interesting points of watching his demonstration was to see how his tank seemed to bounce between walls and moved from one wall to another (i.e. a side to side action), in addition it ran into walls when it needed to turn more tightly. Other than that I did think that their overall approach for running the decoy was a bit better than ours and it completes in a faster manner than ours (in the case where it succeeds, at least).

Appendix A

Ruby Code

See accompanying archive file (*lab4.tar.gz*) for a complete listing of this lab's Ruby and C code.