Joel Van Sickel
5-8-2007
IST 597 F

Final Project

Modeling a Human dTank Commander

A model of a tank operator for the dTank environment is developed for the Soar cognitive architecture using the GUI development tool Herbal. This model addresses modeling more human tank movement. Some design patterns for Soar become apparent and are addressed as well as some discussion on the benefits and shortcomings of Soar, Herbal, and dTank.

Introduction

My primary interest in taking this course was to be introduced to the issue of modeling human behavior. I work in power systems where most avoidable faults are caused by human error. For the final project, I have developed a Soar model in an attempt to more accurately model human behavior in the dTank environment.

Herbal is a tool that can be used to simplify the development of Soar models. It handles many book keeping issues as well as removing the need for a strong understanding of proper syntax used in Soar programs. It is however, still important to understand the inner workings of the Soar architecture, as Herbal will not prevent the development of models that do not work.

Soar is a cognitive architecture that was used for this project because it is the only cognitive architecture I have any knowledge of.

Goals

I undertook three main goals in this project toward making a more human dTank Soar model. I used myself as the human model to attempt to emulate. I played the dTank game numerous times until my strategy stopped evolving and then developed a Soar model inspired by what I learned. This is a poor method (as I have knowledge of the Soar architecture and dTank environment that a normal user would not have), but it suits my task for a class project and saves me time. The goals were to make a tank that made logical decisions, observed its own operation to make sure it did was it wanted to do, and followed a search pattern similar to my own.

My first goal was to develop a tank that only made logical choices in what actions to pursue. This was inspired because once I was good at the game, I would only issue a command that looked like it would accomplish something. This means I would not turn my moving tank into a wall, or attempt to look in a direction I was already looking in. This is a very simple task and involves specifying all of the conditions that must be true for an action to be taken. The conditions for the operators is provided below.

The turnRight operator requires that the current tank heading is not already facing right, and it requires that the terrain to the right be of the type: grass, woods, or road.

Due to the fact that Soar does not allow disjunction with relational operators, and that the tank heading was not an integer (a not equals operator will not necessarily work), I created two turnRight operators, turnRight1 and turnRight2. The operator turnRight1 required that the current tank heading be less than 80 degrees, while turnRight2 required that the current tank heading be greater than 100 degrees. It should be noted that herbal allows disjunction with relational operators and the error will not be detectable until loaded in the Soar debugger or similar environment.

The exact same logic was used for the turnLeft, turnUp, and turnDown operators. Additionally, turnUpRight, turnUpLeft, turnDownRight, and turnDownLeft operators were created. These operators were the same, except for the conditions; it is required that both directions in the operator name have to be of the type grass, woods, or road.

A very similar process was used for look operators. In addition, the tank could look in the direction of low_hills.

My second goal was to create a tank which observed its environment to make sure that it had actually done what it thought it had done. This required the tank to maintain its actual state as well as the state it expected to be in, and compare them.

To do this, I used working memory elements under desiredStates as well as a flag for each working memory element. When the tank issues a command, it changes the value of whatever attribute it intended to change in desiredStates and toggles the flag so that it knows which state to look for a change in.

My third goal was to create a tank that used a more human search pattern. This involved having the tank orient itself at the beginning of the simulation, and then wandering from one side to the other side (pseudo map walking) which is how I searched the map. This required the tank to recognize when it had hit a wall and start searching in a different direction.

It should be noted that everything done was implemented with regard to the tank heading for movement, but not for the turret heading because the concepts are the same, and it is hard to observe the turret heading when running dTank. Before comparing this model to other models, this would have to be rectified.

An additional goal that was not successful was the capability to determine whether the tank tread is damaged of myself or the enemy tank. Because of the lag in when external parameters are updated, it is very hard to know when the position values should or should not change, and I was unable to add this functionality beyond saving past states. I failed to address the issue of how the tank knows when these states should change and whether it is because of an obstacle or damaged tread. This is also an extremely hard situation to test as there is no way to force the enemy tank to damage your tread.

Specifics of the Model

In this section I will cover the set up of the problem spaces and operators with a higher level discussion on them. Specifics on the conditions and actions behind the operators can be found in the appendix in the Soar code or the xml files from Herbal.

Problem Spaces

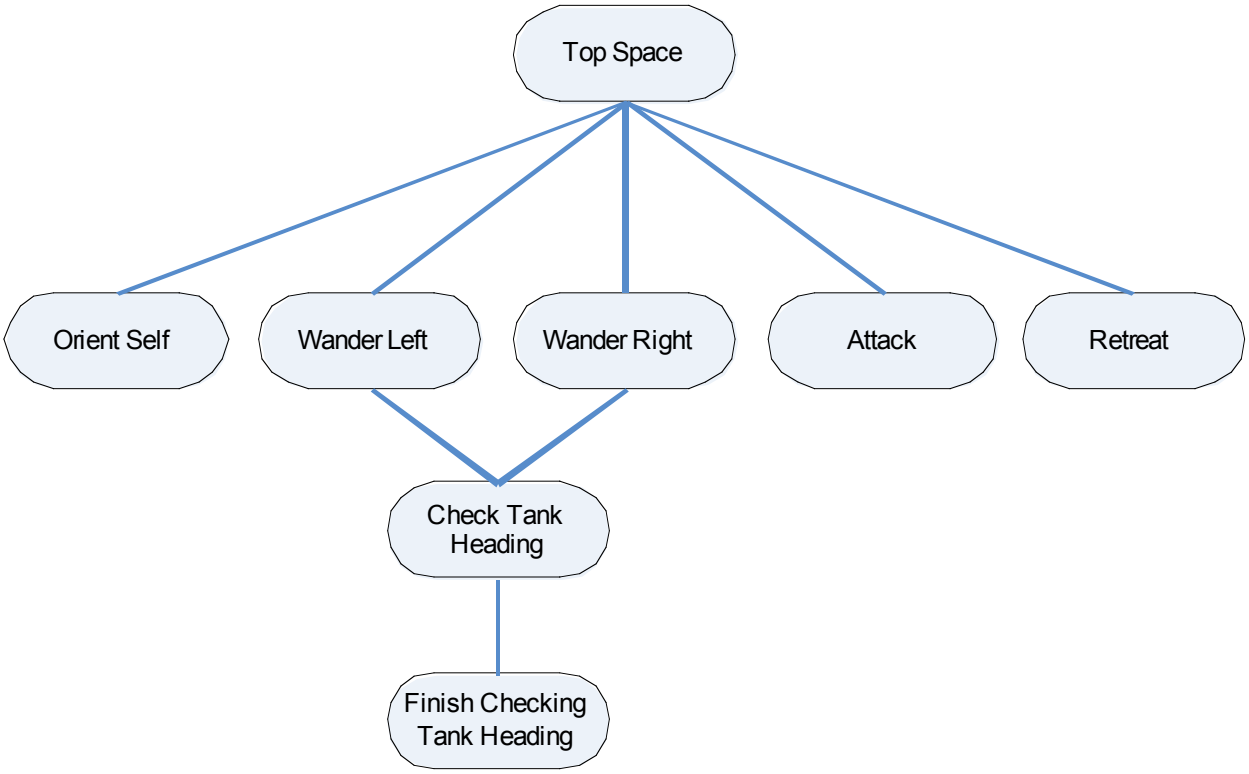

Figure 1. Topology of Problem Spaces

Top Space:

Initial Actions:
- initializeDesiredStates
- initializeCheckTankHeading
- initializeCheckTankHeadingDifference
- initializeDesiredStateDifference
- simpleTurnLeft

This problem space serves to initialize different values that other problem spaces use and turn the tank left which is not necessary but made testing easier.

Orient Self:

Conditions:
- notOriented

Operators:
- orientSelfLeft
- orientSelfRight

This problem space initializes the tank with the correct direction to wander in based on its beginning location.

Wander Left:

Initial Actions:
- move

Conditions:
- wanderingLeft
- noEnemy
- healthy
- oriented

Operators:
- turnLeft1
- turnLeft2
- turnUpLeft1
- turnUpleft2
- turnDownLeft1
- turnDownLeft2
- onLeftWall
- leavingLeftWall

Wander left wanders to the left of the map. There are multiple turn operators because Soar does not support disjunctions. The onLeftWall operator has the tank realize that it needs to turn around, so it is actually a fancy turn right command. The leavingLeftWall operator tells the tank to leave the left wall and switch wander direction, so it is a fancy version of move (some flags have to be set and toggled to ensure the tank actually leaves the wall)

Wander Right: See wander left and replace left with right

CheckTankHeading:

Operators:
- checkTankHeading
- finishCheckingTankHeading

Conditions
- checkTankHeadingFlagSet

This problem space toggles between the two operators until the tank heading becomes what is desired. CheckTankHeading computes the difference between the desired state and the actual state of the tank. FinishCheckingTankHeading orders the tank to change its actual

position to its desired position and then returns back to checkTankHeading to see if that was accomplished.

FinishCheckTankHeading:

Operators:
- clearCheckTankHeadingFlags

Conditions
- closeToDesired

This problem space is entered into when the difference between the desired tank heading and actual tank heading is small. The operator clearCheckTankHeadingFlags then clears all of the flags so that both FinishCheckTankHeading and CheckTankHeading problem spaces are exited back into a wander problem space.

Attack and Retreat:

These problem spaces are not discussed as they have not been modified for the purposes of this project, but are included because of the default settings.

Verification of the Model

This section shows how the model has achieved the desired goals. For the trace shown, an arbitrary example was used where the tank starts moving left, and can only choose between moving up, down, or left to simplify showing the achievement of goals 1 and 2 (right was omitted so that the tank did not immediately turn toward the wall which it does not necessarily start next to).

On the page 7 is a trace of the tank turning up, while observing its state to determine if it was successful. Additionally, all operators fired and transitions into new problem spaces have been bolded.

The first goal was to ensure that the tank only proposed actions it is capable of doing. This has been tested and it works, an example can be seen on the next page on Lines 7 and Lines 8 where the tank is heading left, and only proposes the turnUp operator and the turnDown2 operator (turnDown1 is not possible because it requires the heading to be < 180, but it is at 270).

Goal 2, where the tank observes itself to determine if it did its own action is shown through the following major lines and in the trace.

Line 4: the tank decides to turn up
Line 9: the tank enters the checkTankHeading problem space
Line 13: the tank decides to check its tank heading
Line 14: the current heading is 270, but the desired heading is 0, try again
Line 17: the tank finishes checking its heading, since it failed, it clears only one flag and sets dtank.tankHeading to the desiredStates.tankHeading value (issues the command to turn up again)

Line 19: this loop continues between checkTankHeading and finishCheckingTankHeading until the tank finally observes a change in its environment
Line 23: the tank checks it heading again this time the heading is 0 which is the desired value
Line 28: the tank enters the finishCheckingTankHeading problem space
Line 32: the tank clears all the flags and values and exits to the original problem space (wander in this case)

The next lines show how the tank decides to turn down this time and then the whole process is started over again.

To show the wander path of the tank, a parsing algorithm was written to extract the x and y values of the tank during a simulation run. For this run, the tank starts on the top right side of the map, so it orients itself on the right and chooses to wander left. It then hits the wall, realizes this, leaves the left wall and then wanders right. The parsing code was written in Matlab and has been included in the appendix.

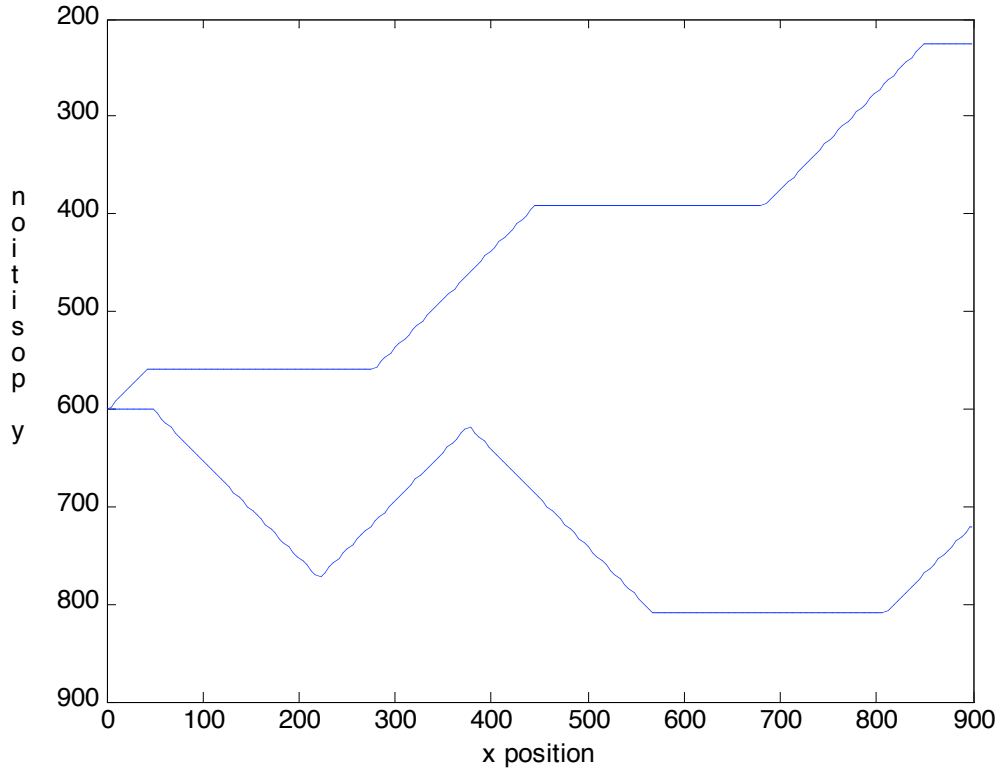

Figure 2: Wander path of the tank

```
Firing |propose*testWander*turnDown2|
Firing |propose*testWander*turnUp|
Retracting |propose*initialize-testWander|
SoarAgent3: Firing |apply*testWander*turnUp|
Turningup
Firing |propose*testWander*impasse*checkTankHeadingps|
Retracting |propose*testWander*turnUp|
Retracting |propose*testWander*turnDown2|
SoarAgent3: Firing |propose*initialize-checkTankHeading|
SoarAgent3: Firing |apply*initialize-checkTankHeading|
Firing |propose*checkTankHeading*checkTankHeading|
Retracting |propose*initialize-checkTankHeading|
SoarAgent3: Firing |apply*checkTankHeading*checkTankHeading|
Computingthedifference: current heading is 270.
Firing |propose*checkTankHeading*finishCheckingTankHeading|
Retracting |propose*checkTankHeading*checkTankHeading|
SoarAgent3: Firing |apply*checkTankHeading*finishCheckingTankHeading|
clearingoneflag
...
Firing |propose*checkTankHeading*checkTankHeading|
Retracting |propose*checkTankHeading*impasse*finishCheckingTankHeadingps|
Retracting |propose*checkTankHeading*finishCheckingTankHeading|
SoarAgent3: Firing |apply*checkTankHeading*checkTankHeading|
Computingthedifference: current heading is 0.
Firing |propose*checkTankHeading*finishCheckingTankHeading|
Firing |propose*checkTankHeading*impasse*finishCheckingTankHeadingps|
Retracting |propose*checkTankHeading*checkTankHeading|
SoarAgent3: Firing |propose*initialize-finishCheckingTankHeading|
SoarAgent3: Firing |apply*initialize-finishCheckingTankHeading|
Firing |propose*finishCheckingTankHeading*clearCheckHeadingFlag|
Retracting |propose*initialize-finishCheckingTankHeading|
SoarAgent3: Firing |apply*finishCheckingTankHeading*clearCheckHeadingFlag|

Clearingtheflags <--this is where the tank returns to wandering-->

Firing |propose*testWander*turnDown1|
Firing |propose*testWander*turnLeft1|
Retracting |propose*testWander*impasse*checkTankHeadingps|
SoarAgent3: Firing |apply*testWander*turnDown1|
turningdown
Firing |propose*testWander*impasse*checkTankHeadingps|
Retracting |propose*testWander*turnLeft1|
Retracting |propose*testWander*turnDown1|
SoarAgent3: Firing |propose*initialize-checkTankHeading|
SoarAgent3: Firing |apply*initialize-checkTankHeading|
Firing |propose*checkTankHeading*checkTankHeading|
Retracting |propose*initialize-checkTankHeading|
SoarAgent3: Firing |apply*checkTankHeading*checkTankHeading|
Computingthedifference: current heading is 0.
```

Design Patterns

This section discusses the main design patterns I have come across in this course.

Repeating an Operator:

An operator cannot typically repeat itself in Soar without an impasse. To get around this, two versions of the same operator can be created that toggle a Boolean value to switch between them.

Disjunctions:

To use an operator when x is < y or > z, two separate operators need to be created, one for each side of the disjunction. Soar can do disjunctions of discrete values, but not continuous spaces. Herbal could possibly take care of this issue under the covers, by automatically creating the two different operators. It is important to note that Herbal will allow the conditions to be created with disjunctions and even generate Soar code, but it will not work.

Ordering Actions

When operators need to fire in a certain order, Boolean flags should be used to know which part of the sequence needs to be fired next. A count can also be used, but the Boolean flags seem more cognitively plausible. This type of behavior is what takes up most of my time when developing Soar agents, as setting up all of the flags and toggling them correctly is quite a hassle, but ordered actions are definitely something that Soar should enable as humans do things in a sequence all of the time.

Funky Numbers

'Funky Numbers' are values that must be stored in working memory to prevent a calculation from putting a value into working memory that already exists there, which deletes the working memory element. Math operations are likely to calculate the same thing twice as some point so it is necessary to avoid this destruction of working memory.

Psuedo Map Walking

A Tank that wanders in the general direction away from the wall, and then switches directions when it reaches the other wall. It is a more realistic way of searching a map, at least when I am the human being looked at. This required ordered action sequences for both sides of the map.

Strict Map Walking

A vacuum that seeks out the top corner of a map and then searches every square. This required ordered action sequences for both sides of the map to move down one space and then start moving the other direction.

Observing State to Determine if Action was Fired

My tank looks at whether it is actually headed in the direction it wants to be headed in, and if not, reissues the order to head in this direction (this does not cause problems, and there are times when the first command did not actually get sent, so the reissue is necessary instead of just

waiting). It takes one operator to compute the difference of the desired heading to the actual heading, and then another 2 operators to deal with whether the difference is too big or not. This requires the ordering of actions, 'funky numbers', and multiple problem spaces.

Summary

This project puts forth a more human like operator of a tank, although, since all capabilities are only implemented on the tank heading and not the turret heading, it is only more human like in the way that it moves, not the way that it searches for an enemy. It does base its movement on its direct surroundings though, so the movement is not decoupled from observing the environment of the tank.

The tank based what wander operators it will choose from based on its surroundings and what it is already doing. It will not try to do what it is already doing nor will it try to do something it knows cannot be done. It observes what it does to make sure it did what it wanted too and tries to do it again if it did not succeed. It also pseudo wanders the map like a human by meandering from one side of the map to other and back.

This model is only somewhat worth taking seriously because there is a lack of good human data to base the action of the tank on or compare it to since I based all action on how I do things, and I can obviously make myself do what my dTank does. I do however show that the tank does what I wanted it to do, and simply getting a Soar model do act the way I wanted it to act is a big deal to me.

I have a few recommendations about Soar, Herbal, and dTank to make based on my experience using them. I think it is ridiculous that Soar removes a value from memory if it is told to put the same value into it. This makes any math with Soar a problem. I had to always set numeric values to weird numbers or there was a chance of my tank losing working memory elements if it performed the same computation twice. This is unacceptable and should be dealt with by the Soar architecture, not the programmer.

Herbal should not be used with Soar unless the basics of Soar are understood (like the problem above and what causes impasses, and also issues like the disjunction) because it does not deal with every little issue. It is a very useful tool though, and saves a lot of time.

dTank has numerous issues that will be resolved over time. My recommendation would be to make the perception factor 1 for the Soar Commander, instead of 100. I think it is supposed to be 10 but no one noticed because no one developed a Soar commander that used knowledge of itself to base its actions on, but that is just a guess. This would make having a model observing itself easier to implement as it would always perceive reality without a time delay. This is an easy change that anyone can make if they have the dTank code building correctly. I developed my dTank model so that it will work with the perception factor of 100.

I also feel that dTank should make a way to observe which direction a tanks turret is facing, as when I tested programs, I could never tell if my tank was looking where I wanted it to look without reading through the log.