TU Berlin
Department of Artificial Intelligence

Projects in Machine Learning and Artificial Intelligence

WS 2010/11

Project Report

Learning to ride a Light Cycle

Joachim Haenicke

Berlin, 30/05/2011

Contents

1 Introduction

1.1 Origin of the game TRON

In 1982, Walt Disney Pictures released the American science fiction movie *TRON*, a movie utilizing a high degree of computer animations for its time. One of the key action features of the movie are the so called Light Cycle matches. The general concept of these originates back to the classic video game *Snake* from the late seventies and has been adapted in numerous video games since.

Although some adaptations tried to mimic the three dimensional design of the Light Cycles and their environment, such as the popular Linux version *Armagetron*, the basic elements of the game's concept can be easily visualized in two dimensions. This way, versions of the game could also be integrated in early mobile phones, reaching millions of people.

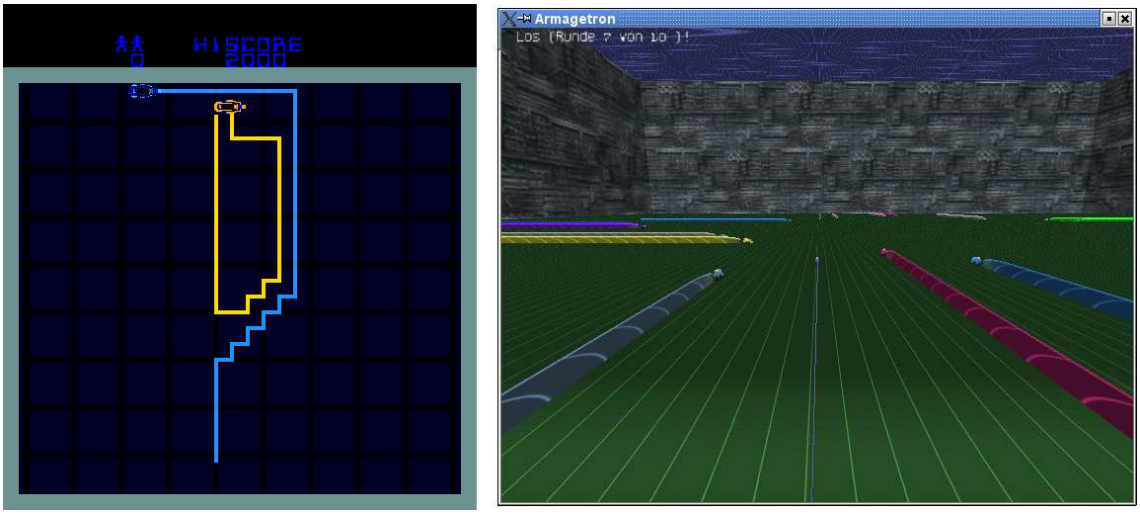

Figure 1: *Two version of TRON: TRON arcade game released in 1982 (left) and Armagetron, running in windows mode (right).*

1.2 Motivation and aim of this project

One specific realization of a snake-like game is the masters' design group's *Ultimate Tron II*, written by Oliver Stiller for the infamous 8-bit home computer *Commodore 64* (C64), the most popular computer during the 1980s. The main screen of the game is shown in figure 2 for reasons of nostalgia, the details of this multi-player game are described in section 2.1. As the game combines this specific set of vital game elements, it enforces a high degree of real-time interaction between players, and together with its nearly infinite replayability it can induce a great amount of pleasure among players, making it possibly the second best game ever to be programmed[1].

[1]The author reluctantly accepts that this statement might be personally biased.

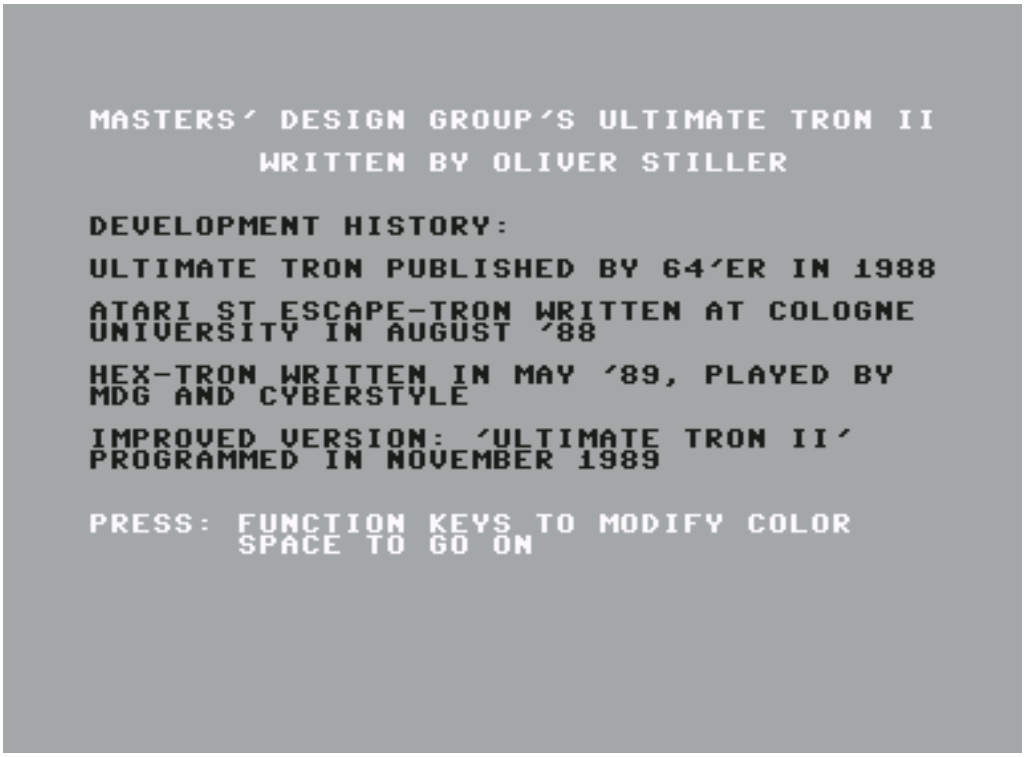

Figure 2: *Ultimate Tron II on the C64: Intro screen.*

Inspired by this classic video game, the author implemented an extended version of the game (see section 2.2). One additional feature of this game is the option of including computer controlled opponents. Most artificial agents that are used in computer games, even in current ones, act on the basis of predefined rules. The aim of this project is to construct a simple but partly autonomous artificial intelligence (AI) that extracts rules for a rational action selection by exploring its environment on the basis of a reinforcement paradigm.

Though the game seems trivial, even the classic version yields problems on many levels of complexity. Taking into account all additional elements of the extended version, constructing an AI that learns its actions can be arbitrarily complicated. Therefore, this very minimalistic and intuitive game is very well suited for the implementation and analysis of AI algorithms of different scales, both without and with self-organizing aspects. This work might also serve as a motivation for other students entering the field of AI and computer games.

2 Game description

2.1 Ultimate Tron II

The typical graphics mode of the C64 generates a screen with a resolution of 320x200 pixels and 16 colors. The programmer of this game has chosen a minimalistic two dimensional visualization consisting of only two colors that can be chosen from all 16 representable colors (see figure 3). The game is designed as a multi-player game for up to $N = 6$ players. Each player controls the movement of a line with only two keys[2]. The two keys change the movement of a line 90 degrees to the left (i.e. counter clockwise) or 90 degrees to the right (i.e. clockwise), respectively.

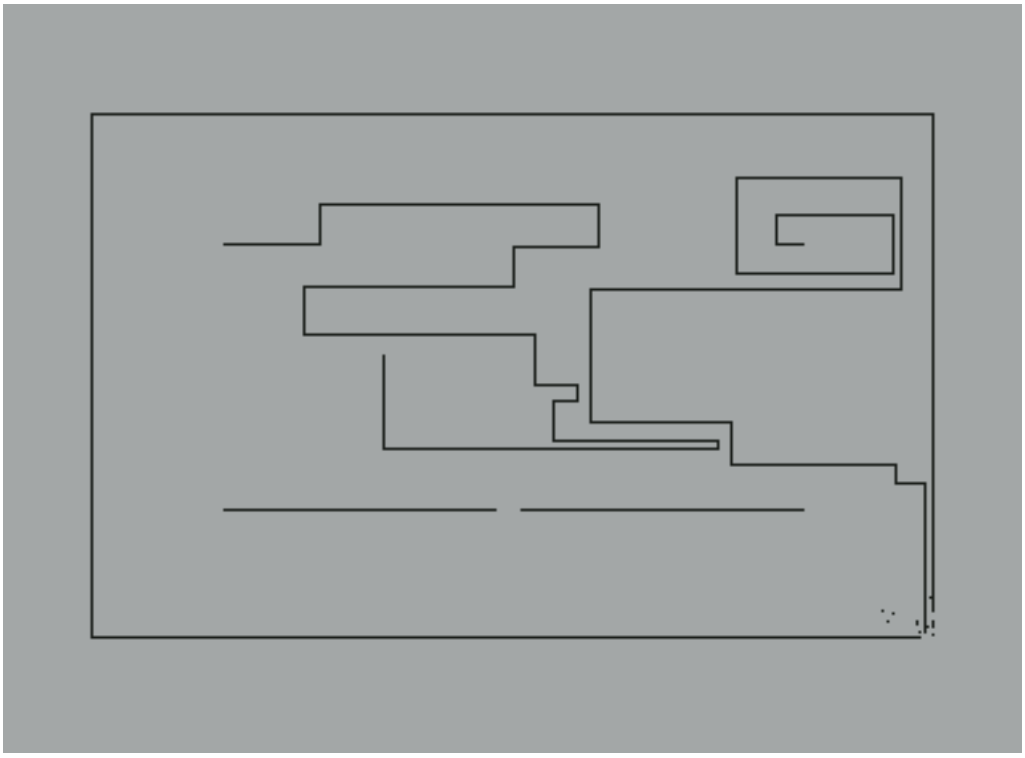

Figure 3: *Ultimate Tron II on the C64: In-game screenshot of a game with four players. The two players that started at the lower positions did not change their original direction and ran into each other. One player started at the upper right position and hit the wall in the lower right corner, producing wholes in the frame. The upper left player did not crash and will win this game.*

At the start of a new game, each of the N player's lines starts at one of N predefined starting positions, with the player's names appearing for a short interval. After the names have disappeared, all lines grow longer along the four main directions (up, left, down, right), depending on the actions chosen by each player, and will eventually hit an obstacle. Obstacles can be other player's lines, one's own line, or the frame. Once a line hits an obstacle, the associated player is out of the game and a few temporarily existing particles[3] appear that act themselves as obstacles and clear

[2] In most snake/nibbles-like versions the player controls the movement with four keys - one for each direction - which sucks.

[3] These particles are also known as Broesel.

their path off any other obstacles. This way holes in the outer frame might appear (see figure 3) allowing any remaining line to escape through these. Once such an escape happens happens, the game ends instantly and the player who escaped wins. Otherwise, the game ends either if one player manages to stay alive for a little longer that all other players, or if all players have crashed. When a game has ended, players collect varying amounts of points (depending on each game's outcome) and a table of the game's statistics is shown. Shortly afterwards, a new game can be started by pressing a button. The amount of games that will be played is only limited by the player's persistence.

2.2 Advanced Ultimate Tron II

On the basis of Ultimate Tron II an extended version was programmed that runs on current PCs[4]. Although it includes many general improvements and additional in-game elements (bonuses with various functions can be picked up, see figure 4), these will not be documented in detail in this work, since the implemented AI-algorithms do not take these into account.

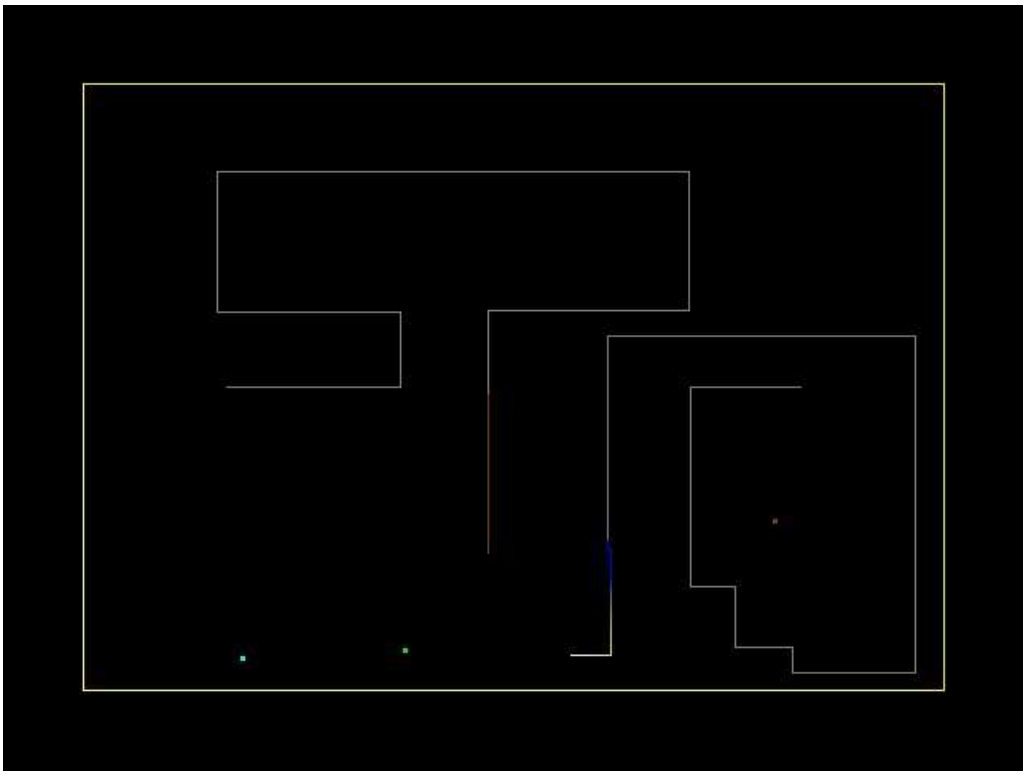

Figure 4: *Extended Ultimate Tron II: In-game screenshot. Two player compete in this game. The left player collected a brown bonus, the right player collected first blue bonus and a yellow bonus shortly afterwards and has activated the yellow bonus. Three other bonuses (cyan, green and brown dots) are waiting to be picked up.*

[4]C++ software was developed in Eclipse, using OpenGL and SFML as additional libraries.

2.3 Game description in terms of AI

Before constructing an algorithm that learns to control the movement of a line in an effective way, i.e. that maps certain percepts to specific actions in order to reach a goal, the game's principles need to be analyzed and all possible percepts, actions, and goals that any agents are supposed to deal with need to be specified. Figure 5 shows a comparison of different agent types with their associated **p**ercepts, **a**ctions, **g**oals, and **e**nvironments (PAGE).

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Taxi driver	Cameras, speedometer, GPS, sonar, microphone	Steer, accelerate, brake, talk to passenger	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers
Line (complete environment)	Bird view of environment, position, speed, direction	Steer, activate bonuses	Human like behavior: Survive, kill and make fun of opponents, escape, collect bonuses	Frame, own line, other lines, bonuses
Line (reduced environment)	View of immediate environment	Steer	Avoid collisions	Frame, own line

Figure 5: *Table with examples of agent types and their different PAGE descriptions (adapted from [Russel 95]).*

Obviously, an optimal[5] artificial opponent would mimic human behavior as closely as possible. But processing all relevant percepts in real time to extract those relevant features that a human player uses within a multi-player game to choose her actions is a complicated task. Exercised human players quickly extract various relevant dynamically changing features of the scene, such as the game states[6], the states[7], positions, and directions of movement of their own and other players' lines, and the positions of bonuses and possible holes in the outer frame. On the basis of these stimuli human players show various strategies in order to choose their actions. In addition, human players exhibit clear limitations, such as reaction time and constrained regions of focused attention within their visual field.

The integration of all of these features into an AI would have by far exceeded the scale of this project. Consequently, all aspects of the game had to be reduced considerably.

[5] Optimal with respect to game play.

[6] In the full version of the game, lines can collect bonuses and activate those, possibly changing their own and/or other players' states, e.g. line position, speed, and visibility.

[7] see previous footnote

To start with the easiest possible scenario, other players and all bonuses we excluded from the problem. This still leaves a problem with a vast state space, as described in section 2.3.1, so percepts were cut down to the readout of three sensors that scan parts of the immediate surrounding of the agent. This, plus the exclusion of all bonuses, led to a crucial reduction in state space. Figure 6 shows the environment of the reduced version of the game as it was used for the rest of the project.

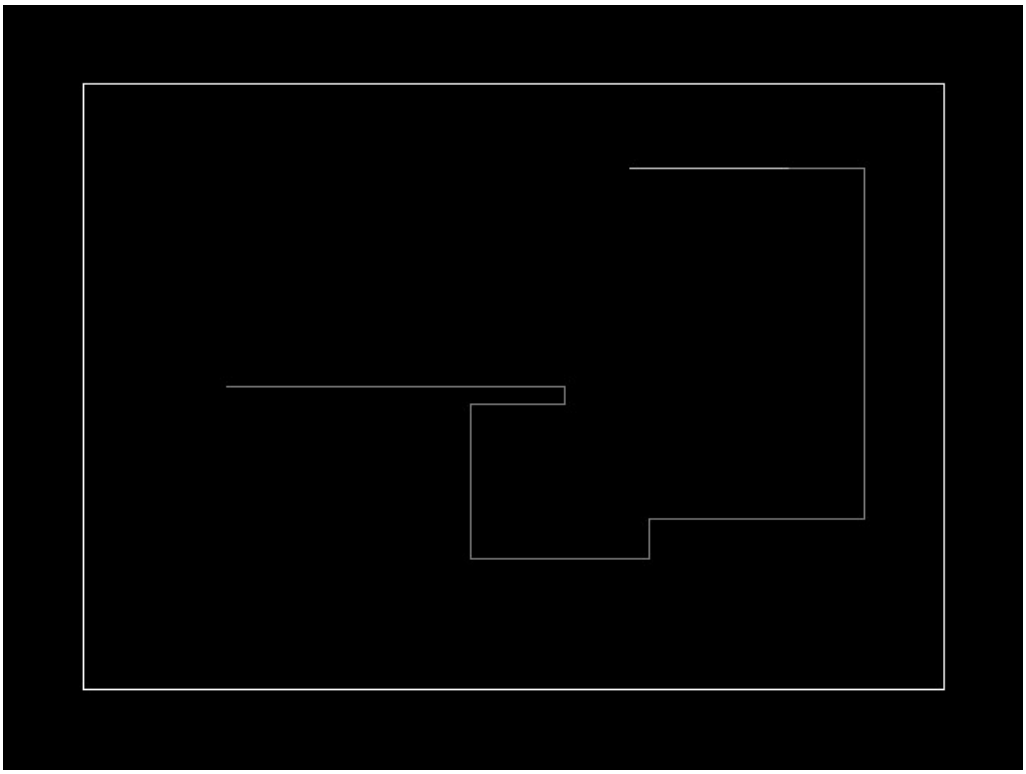

Figure 6: *The reduced version of the game: A single agent navigates in a deprived environment.*

Figure 7 lists the properties of different environments and might give an idea of how difficult the problem of the complete game is. As noted in [Russel 95] "answers can change depending on how you conceptualize the environment and agents".

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Medical diagnosis system	No	No	No	No	No
Part-picking robot	No	No	Yes	No	No
Taxi driver	No	No	No	No	No
Tron (complete)	No	No	No	No	Yes
Tron (reduced)	No	Yes	No	Yes	Yes

Figure 7: *Table with examples of environments and their characteristics. (adapted from [Russel 95]).*

2.3.1 The reduced state space

A standard example for many of the described algorithms in [Russel 95] is the traditional grid world problem. A similar description of a region of pixels around the end position of a line, say a 10x10 square, would yield a possible state space with a vast number of elements (in a general setting nearly 2^{100} possible states), even without any opponents and other complicating parts in the game (bonuses, escaping through holes in the outer wall). This is why a reduction of the state space was crucial. Still, one can think of any number of reduced representations of the agent's environment that pickup some of the relevant features. To get to an easily implementable solution, different simple versions of a reduced state space were tested. Out of these, the one shown in figure 8 proved to be most useful.

Three virtual sensors scan the environment to the left, to the front, and to the right of the agent, respectively. Each sensor is defined by a number of different states N_s, a range r given in number of pixels, and the range of $N_s - 1$ subregions, also given in number of pixels. If there is no obstacle within a scanners range of sight, the scanner will be in state $s = 0$. The region of sight of each scanner is divided into $N_s - 1$ subregions, which are signified for an example of $N_s = 4$ by the colors green, yellow, and red in figure 8. If there are any obstacles within sensor sight, the one closest to the agent's current position will determine the sensor state to be $1 \leq s < N_s$, accordingly. All AI algorithms that were implemented within this project were based on such a sensory state space with $r = 45$ and $N_s = 5$ for a total of 501 possible states (sensory states, availability of actions, terminal state). Ranges of the four subregions (from closest to farthest) were chosen to be 2, 4, 8, and 31.

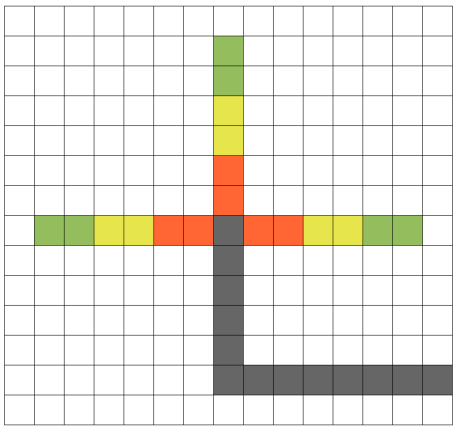

Figure 8: *The reduced state space: Three distance sensors provide the agent with discretized information about possible obstacles. Gray squares: The agent's line. Green, yellow, and red squares: Regions of low, medium, and high sensor activity, respectively. White squares: Empty environment.*

It is important to mention that with such a reduced state space, the original problem of learning appropriate actions in the full environment has been reduced immensely to the problem of learning appropriate actions within this very limited feature space,

which might be considered a trivial task. In contrast, this simplicity might also serve to make it an easily graspable example.

3 Implemented AI algorithms

In a first step, to prove the efficiency of the reduced state space, a simple rule based version of an AI was implemented. Subsequently, a so-called Q-learning agent as described in [Russell 95], section 20.6 was constructed and tested with different parameters. This section gives a detailed description of these two algorithms, section 4 shows some selected results for both of them.

3.1 A simple rule-based AI

The behavior of this AI is determined by the following two straight forward rules:

- If one of the side sensors is more active than the front sensor, the next action will be turning into the direction of the less active sensor.
- If both side sensors are equally active and more active than the front sensor, the next action will be turning into a randomly chosen direction.

Two additional mechanisms we added, that slightly complicate the agent's behavior. One of them accounts for the limitation in speed that human players show when choosing the same action twice, i.e. pressing one of the action keys twice: once the AI has turned into one direction it can execute the same action again only after a randomly chosen interval of 5 to 15 steps[8].

The other mechanism introduces some randomness into the agents actions: The agent executes its next action only if a random integer in the range of 0 to $N_s - 1$ is smaller than the current front sensor's state. Thus, the higher the activity of the front sensor, the more probable it is that the agent executes its next action. If the front sensor is in its highest state, i.e. if there is an obstacle directly in front of the line, the agent will always execute its next action. Figure 9 shows the C++ source code for this algorithm.

[8] After each step, a line moves one pixel into its current direction.

```
void C_AI_sense_O_matic::sense_O_matic()
{
   if ((sensors[0]>sensors[1])||(sensors[0]>sensors[2]))
   {
      if (sensors[1]>sensors[2])
      {
         if (!a1)
            next_action = 1;
      }
      else if (sensors[2]>sensors[1])
      {
         if (!a2)
            next_action = 2;
      }
      else
      {
         int i_buffer = game->rand_int(0,1);
         if (action_available[i_buffer]<=0)
            next_action = i_buffer+1;
         else if (action_available[1-i_buffer]<=0)
            next_action = 2-i_buffer;
      }
   }
   if (game->rand_int(0,4)<sensors[0])
   {
      action = next_action;
      action_available[next_action-1] = RT;
      RT = game->rand_int(5,15);
   }
}
```

Figure 9: *C++ source code of the simple rule-based AI. Front, left, and right sensors are addressed by sensors*[0]*, sensors*[1] *and sensors*[2]*, respectively. Actions left and right were assigned to values 1 and 2, respectively.*

3.2 An exploratory utility-learning agent

For the implementation of this AI, the algorithm taken from [Russel 95] (see figure 10) was slightly adapted. At the basis of this algorithms a utility (also called Q-value) gets associated to each state-action pair the agent experiences, and through the temporal difference (TD) learning rule, successive experiences (state-action pairs) get connected with each other, effectively propagating utilities through the state-action space:

At first, when a naive agent is in a specific state i, all state-actions pairs $[i, a_1]$, $[i, a_2]$, ..., $[i, a_{N_a}]$ will have the same utility: $Q[i, a_1] = Q[i, a_2] = ... = Q[i, a_{N_a}] = 0$, with N_a being the number of possible actions. In this case, a random action a_x will be chosen. Afterwards, the agent is in a new state $[j]$ and needs to get a feedback $R(i)$ about the utility of the previous state-action pair. This value can also be seen as a reward for positive values or as a punishment for negative values. With this

feedback, the agent updates its utility $Q[i, a_x]$ according to the following equation.

$$Q[a, i] = Q[a, i] + \alpha \left(R(i) + \max_{a'} Q[a', j] - Q[a, i] \right) \tag{1}$$

Here, α is a learning rate and $\max_{a'} Q[a', j]$ is the highest utility for all possible actions of state j based on the agent's current knowledge. The next action is then determined with the help of a so-called exploratory function $f(\max_{a'} Q[a', j], N[a', j])$, with $N[a', j]$ being the number of times action a' has been executed in state j. All results presented in chapter 4 were produced using $\alpha = 0.5$ and the following exploratory function $f(U, N)$:

$$f(U, N) = \begin{cases} 1 & \text{if } N < 3 \\ U & \text{otherwise} \end{cases} \tag{2}$$

This algorithm is shown in detail in figure 10. As mentioned before, it needed to be adapted to the nature of the problem at hand, namely that in all scenarios terminal states produced the most important feedback in the form of a negative reward. To this end, the algorithm was changed so that a reward/punishment is also received if j is a terminal state. Additionally, equation 1 is calculated once more when the agent has reached a terminal state, so that the punishment in the terminal state gets associated with the previous state-action pair that led to this state.

```
function Q-Learning-Agent(e) returns an action
    static: Q, a table of action values
            N, a table of state-action frequencies
            a, the last action taken
            i, the previous state visited
            r, the reward received in state i

j ← State[e]
if i is non-null then
    N[a,i] ← N[a,i] + 1
    Q[a,i] ← Q[a,i] + α(r + max_a Q[a',j] – Q[a,i])
if Terminal?[e] then
    i ← null
else
    i ← j
    r ← Reward[e]
a ← arg max_a f(Q[a',j],N[a',j])
return a
```

Figure 10: *An exploratory Q-learning agent, taken from [Russell 95]: "It is an active learner that learns the value $Q(a, i)$ of each action in each situation". $f()$ is an exploratory function that defines how curious the agent is. This algorithm does not learn anything if only terminal states are rewarded/punished.*

4 Results

4.1 Behaviour of the simple AI

Figure 11 shows the behavior of the simple rule-based AI. As can be expected from the rules, the AI changes direction once an obstacle comes into the range of the front sensor. Randomness in the execution probability of each action produces a slightly inhomogeneous trace.

The agent's overall behavior is good enough to make it an opponent of intermediate difficulty for human players when used in a multi-player scenario. It can also serve as a comparison for any other algorithms.

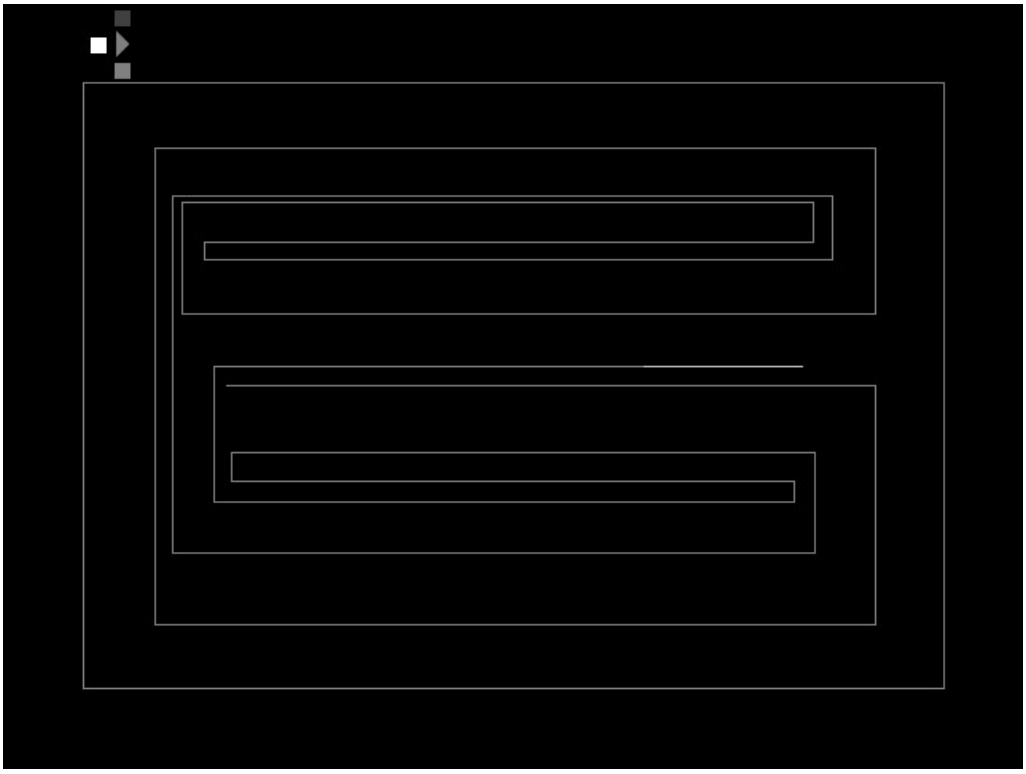

Figure 11: *An exemplary trace of an agent acting according to the simple rule-based AI. The light gray part of the line shows its front. Symbols on the top left: current direction of the agent (triangle) and sensory read-out (sqares). Color of squares shows sensory state, white for highest activity, black for no activity, scales shades of gray for intermediate activity.*

4.2 Behaviour of the Q-learner

Unfortunately, only a very limited amount of time was afforded to characterize and explore the parameter space of the Q-learning algorithm. Nevertheless, figures 12, 13, and 14 show some exemplary traces of varying sets of parameters.

In all cases the reward in the terminal state was $R(T) = -12$. If the agent gets no other feedback on its state-action pairs, it will always show a random walk as in figure 12. This results is due to the fact that the agent will only learn to propagate a negative reward along to those states which always led to a pre-terminal state. Since any pre-terminal state can also lead to a non-terminal state, the most optimistic guess will be always a Q-value of zero, so that all states prior to a pre-terminal state won't change their Q-value. In addition, the agent cannot differentiate between the two last states before any terminal state on the basis of the current sensor design.

If one includes just a small variation, namely setting a reward for no action taken, $R(\neg A)$, the agent starts learning to avoid obstacles within its sensory range, still showing random walk behavior if there is no sensory input, as is illustrated in figure 13. This can be avoided by adding to the algorithm of the Q-learning agent the simple rule of not executing any actions if all Q-values are equal. As a result, the agent shows a very stereotypical behavior already after 30 to 50 games (see figure 14). This behavior leads to one of the longest possible traces and therefore might be described as a near optimal behavior within this single-player framework.

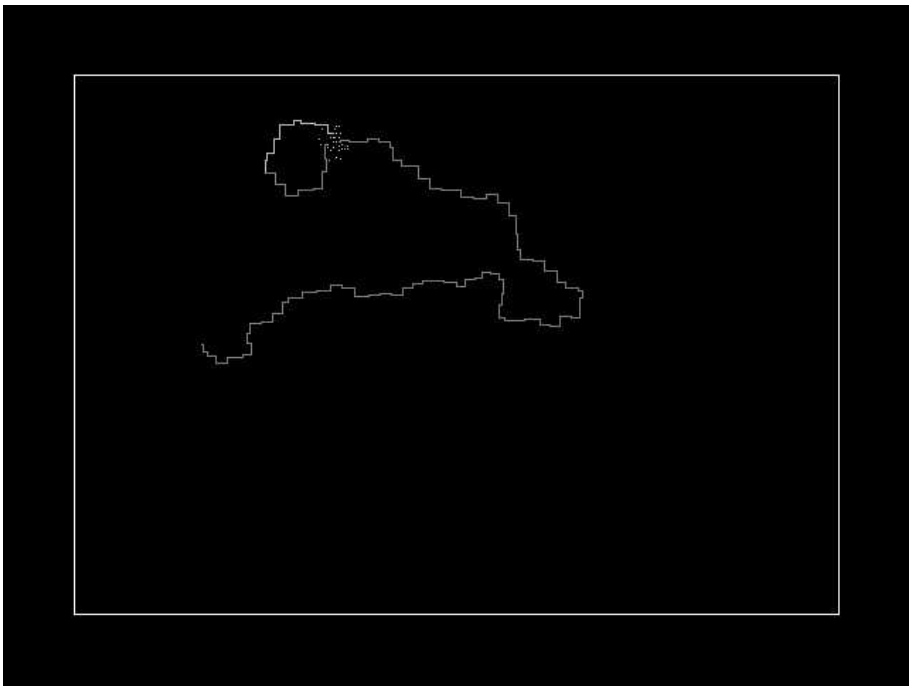

Figure 12: *An exemplary trace of an agent acting on the basis of the Q-learning algorithm. The light gray part of the line shows its front.*

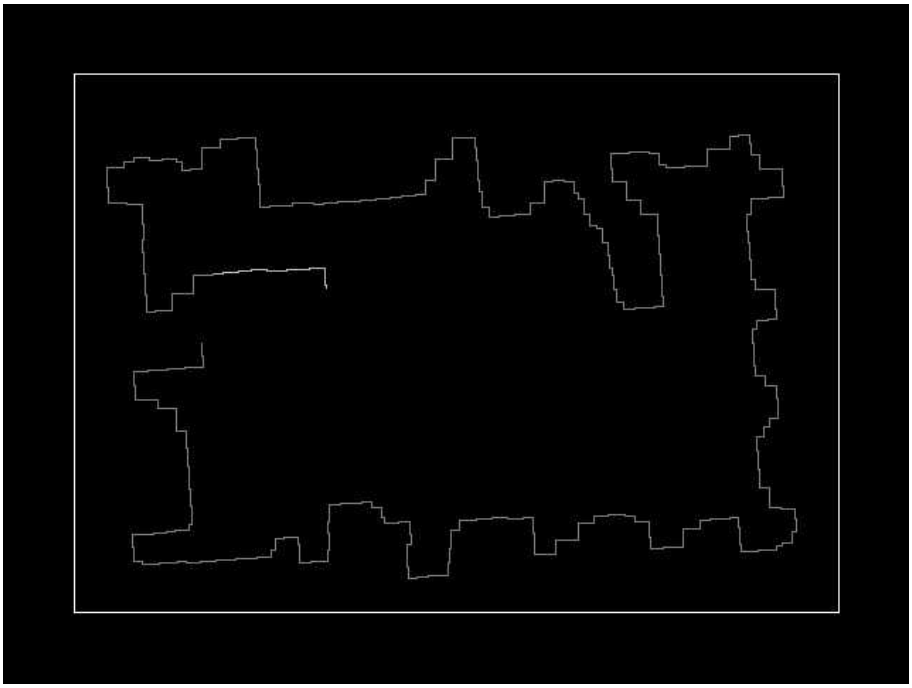

Figure 13: *An exemplary trace of an agent acting on the basis of the Q-learning algorithm after approx. 200 games.Though the agent has learned to avoid obstacles it still shows sections of random walks. The light gray part of the line shows its front.* $R(T) = -12$, $R(\neg A) = 0.01$.

Figure 14: *An exemplary trace of an agent acting on the basis of the Q-learning algorithm. The light gray part of the line shows its front.* $R(T) = -12$, $R(\neg A) = 0.01$. *Additional rule: when all Q-values are equal, no action is taken.*

5 Discussion

5.1 Summary

In the course of this project, the following steps we taken in order to construct a simple but partly autonomous artificial agent:

- The original problem of learning a complex multi-player game was reduced to a much simpler single-player approach without additional game elements.
- The original high dimensional input space was reduced to a much smaller sensory state space.
- A rule-based AI was implemented and tested using the reduced state space.
- A reward-based algorithm that learns to associate Q-values with state-action pairs was implemented and tested using the reduced state space.

Even though it is a very simple algorithm, there are lots of parameters and rules that can be tweaked in order to optimize the agents behavior towards specific goals. And although the implemented Q-learning agent was characterized only for a very small subset of its parameter space, it could be shown that the agent can learn obstacle avoidance. Additionally, small changes within the algorithm can lead to many interesting, sometimes non-intuitive behaviors.

Some shortcomings of the algorithm were identified in the course of testing. Although these will need to be analyzed in more detail, it is apparent that when giving (negative) rewards only in terminal states, the agent cannot learn these, because they won't be propagated along to earlier states. This erroneous property of the algorithm needs to be fixed in the near future by propagating experiences further into the past. In addition, many other possible improvements can easily be conceived:

- The number of sensor states and/or the number of sensors could be increased. This would enlarge the state space but also give room for more intricate behavior.
- New kinds of sensors could be implemented that might extract some more of the relevant features.
- By adding abstract features to the state space, such as the distance to the nearest bonus, more complex elements of the game could be accounted for.
- Different algorithms could be tested and evaluated against each other within a multi-player setting.

This list could be continued without great effort, since the game elements of Advanced Ultimate Tron II yield arbitrarily complicated problems for the construction of an AI. This work represents only a small step towards the goal of programming an autonomous human-like artificial agent for this simple yet fascinating game.

References

[Russell 95] Russell, Stuart; Norvig, Peter:
Artificial intelligence: a modern approach
Prentice Hall (1995)