# CSCI1410 Fall 2020
# Final Project: Tron-141

Project Due Monday, December 14 at 11:59pm ET

**December 14 is a hard deadline.**
**No late projects will be graded, and**
**no late days can be used for the final project.**

## 1 Important Dates

| Milestones | Date | Time |
|---|---|---|
| Partner Form Due | 11/25 | 11:59am ET |
| Warm-up Bot & Writeup Due | 12/7 | 5:59pm ET |
| Mentor TA Meeting Deadline | 12/7 | 5:59pm ET |
| Tournament Begins | 12/7 | 11:59pm ET |
| Final Bot & Writeup Due | 12/14 | 11:59pm ET |
| Tournament Ends | 12/15 | 11:59pm ET |

**Note:** If you are taking this course as a Capstone, you must work individually. Otherwise, you are encouraged—but not required—to work in pairs. You may choose your own partner, or ask us for help finding a partner. To request our help, it is best to communicate with us via Piazza.

Either way, **you are required to submit a Google form informing us of your plans by 11/25 at 11:59am ET.** Do not miss this deadline. We cannot assign you a Mentor TA until you submit this form, and the sooner you are assigned a Mentor TA, the sooner you can get started in earnest on the project.

## 2 Goals

The goal of this final project is to help you synthesize all the AI knowledge you have garnered over the course of the semester. The project entails building an AI bot to play a grid game (i.e., a game played on a discrete grid). To build an effective bot for this task will require that you employ multiple techniques, ranging from adversarial search to machine learning (i.e., function approximation) to reinforcement learning to multi-armed bandits. The best bots will utilize all of these techniques and more.

## 3 The Game

The game of Tron derives from a Disney movie of the same name. In the game, two motorcycles (agents) drive around a wall-enclosed grid at a constant speed. The motorcycles can continue in the direction they are going, turn right, or turn left. Whenever they exit a cell, they leave behind an impenetrable barrier, which makes that cell uninhabitable. Eventually, one of the two agents has no choice but to crash into a wall or a barrier. At that point, the game ends with the agent who crashed as the loser.

As studied in the AI literature (e.g., [1, 2, 3], Tron is a two-player, simultaneous-move, zero-sum game. Tron-141, however, which is the game we have designed as the final project for CSCI 1410, is a sequential-move game, with the two agents moving in turn. Many Tron agents built for the simultaneous-move version use heuristic strategies that incorporate aspects of adversarial search (e.g., $\alpha\beta$ pruning), making them readily applicable to alternating-move games. We have eliminated some of the original game's complexity by changing the rules in an arguable unnatural way such that only one motorcycle can drive at a time.

## 3.1 The Rules

Tron-141 is a two-player, alternating-move, zero-sum game played on a walled-in rectangular grid (i.e., the board), in which players take turns moving straight ahead, left, or right, leaving behind an impenetrable barrier. A player loses by colliding with a barrier or a wall.

Below are two example 7x7 game boards. The one on the left is the initial board (also called a **map**), and the one on the right is the same board after Player 1 has moved down and Player 2 has moved up.

```
#######
#1    #
#     #
#     #
#     #
#    2#
#######
```

```
#######
#x    #
#1    #
#     #
#    2#
#    x#
#######
```

The numbers 1 and 2 denote the current locations of Players 1 and 2, respectively; the `#` symbols denote permanent walls; and the `x` symbols represent the barriers that the players have left behind.

Below are two example 13x13 game boards. These boards are initialized with additional walls, beyond those enclosing the board. Note that the players initial positions can vary.

```
###############
#             #
#  ##         #
#  ## 1       #
#     ###     #
#     ###     #
#       2 ##  #
#         ##  #
#             #
###############
```

```
###############
#             #
#    1        #
#    #####    #
#    #   #    #
#    #   #    #
#    #####    #
#        2    #
#             #
###############
```

**Note:** Tron-141, in its full generality, also includes powerups. These powerups are present in the code, and explained in Appendix A. While it may be fun for you to tailor your AI bot to powerups, this project is sufficiently rich without them, so you need not pay them any mind.

**Another Note:** To get a feel for how the game works, we recommend you run through a few example games. To do so, run `gamerunner.py` without any command-line arguments. This code will execute a game between two bots who choose their moves randomly, and print the stream of boards to your terminal.

## 3.2 Time Limit

Each player must move within **1 second**. If a player does not move within this time frame, the simulator moves them Up. Furthermore, we cannot accept multithreaded bots, because code that inadvertently is not thread safe could jeopardize our class tournaments.

### 3.3 Evaluation

We will evaluate the success of your AI bot in several ways. We will evaluate your core ideas on the basis of your writeup. We will also test your bot's performance against various TA bots. Finally, during finals week and beyond, we will run a tournament, where all the Tron-141 bots developed this year will compete against one another. Your grade will be a combination of your successes along all of these dimensions.

## 4 Approaches

About a decade ago, solving Tron was an ongoing research challenge. Here are links to a few papers chock full of ideas that should be useful as you work on this project:

- Endgame Detection in Tron
- A UCT Agent for Tron: Initial investigations
- Monte-Carlo Tree Search for the Simultaneous-Move Game Tron
- Monte Carlo Tree Search for Simultaneous-Move Games: A Case Study in the Game of Tron[1]
- Google even ran a Tron competition; one competitor's post mortem is posted here.

### 4.1 Adversarial Search

The most basic approach to building an AI bot to play a game is to simply hard-code some reasonable heuristic behavior. A popular heuristic for Tron is the *wall-following* heuristic, which favors moving along the walls. This heuristic is implemented in one of the weaker TA bots, but note that wall-following is not a terrible idea if your opponent is also a wall-follower, especially if the games were scored according to how long they last. An alternative to wall-following is the *path-planning* heuristic, in which a player searches for a move that would afford it the longest continuation: i.e., the longest available path following said move.

A more principled approach to solving any two-player, alternating-move, zero-sum game is to use the minimax algorithm. A more efficient yet equally principled approach (since it is provably equivalent to minimax in terms of the solutions its finds) is to use $\alpha\beta$-pruning, which can prune nodes in the search tree. But even $\alpha\beta$-pruning is not efficient enough to solve Tron, without artificially inhibiting the depth of its search. Consequently, heuristic evaluation functions that incorporate domain knowledge are required, much like they were for Connect Four, in the Adversarial Search homework assignment.

The aforementioned papers outline several heuristic evaluation functions, including those implemented in the TA bots. Most, if not all, the heuristics employ some form of "space estimation," because it is advantageous for a player to be surrounded by free space, rather than walls and barriers. There are various ways to estimate space. One of the more naive approaches simply counts the number of contiguous cells accessible to a player,[2] while a more sophisticated variant counts the length of the longest paths in this region, since it is not possible to traverse all free cells. But neither of these approaches pay any mind to the opponent—who is vying for the very same space! So what makes more sense in this game is to compute a (naive or sophisticated) space estimate for *both* players, and then to define a heuristic evaluation function that combines these estimates in some way (e.g., take their difference, their ratio, etc.), as it is these estimates combined that contains the most information about which player is *en route* to winning the game.

An alternative heuristic evaluation function which bakes in consideration of the opponent is described (with pictures!) in Endgame Detection in Tron. This approach labels each cell as closer to either Player 1 or 2, as measured via Manhattan distance. These labelled regions are called *Voronoi regions*, while the cells that are equidistant to both players are called the "battlefront." As above, the heuristic value is then the difference in the sizes of the two players' Voronoi regions, thereby predicting the winner to be whichever player's Voronoi region is larger. This heuristic evaluation function is sensible because the player with the larger Voronoi region can win the game by moving directly to the battlefront and then proceeding to cordon off their Voronoi region. `TA-Bot2`, the best of the TA bots, uses $\alpha\beta$-pruning with this Voronoi heuristic.

[1] The last two papers have overlapping authors, and hence, quite likely, overlapping ideas.

[2] `TA-Bot1` uses $\alpha\beta$-pruning with a heuristic that estimates a player's free space.

## 4.2 Reinforcement Learning

As we discussed in class, state-of-the-art methods for AI game playing, like AlphaGo [4], employ reinforcement learning (RL) methods. Recall that RL is applicable in Markov decision processes (MDPs). Hence, to employ RL in a game requires that we view the game as an MDP. This reduction is almost entirely straightforward: the states in the MDP are the board configurations (i.e., the nodes in an adversarial tree search); the actions in the MDP are the players' available moves at each board configuration; and the rewards at the terminal states indicate who the winner is. Only the transition probabilities are potentially ill-defined. But given an opponent strategy (e.g., a wall-following agent), their behavior defines the transition probabilities. Moreover, so long as that opponent's strategy is stationary—meaning the distribution over actions it employs depends only on the state but not on time—the Markov property is satisfied.

**N.B.** Whereas an adversarial search tree models *both* players, so that nodes are labelled with players' identities indicating whose turn it is to move, the aforementioned MDP models only one player, so that every state corresponds to just one player's actions. The other player's actions are folded into the transition probabilities; they are not modelled explicitly.

Given this reduction, it is straightforward in principle to learn to play the game of Tron (or even the game of Go) using RL. Well, not so fast! Just as $\alpha\beta$-pruning is intractable in Tron (and Go) if it is not depth-limited (i.e., it is impossible to visit all nodes in the search tree), it is likewise impossible to evaluate all states in this MDP. However, it is not intractable to evaluate a few states. Hence, the way to use RL in game-playing is to combine it with supervised learning (*a.k.a.* function approximation), so that we can learn the values of a few states, and then generalize those values across many states. As usual, we can represent states (i.e., board configurations) in terms of their features, use non-linear basis functions to transform those features into a richer "derived" feature space, and then regress, either in a batch fashion using least squares, or incrementally, using stochastic gradient descent, to learn a value function for the MDP.[3]

In principle, if we could visit a small but representative sample of the game states, it is conceivable that we could create a data set consisting of precisely those representative game states and their estimated values, from which we could generalize effectively to all the game states. But who is to say a small, representative sample even exists? Here is another, related, idea. What if instead of trying to learn the value function, we instead tried to learn an optimal policy outright, via an algorithm like policy iteration? Just as we are unlikely to be able to visit enough states to learn the value function exactly, it is also likely impossible to learn an optimal policy at *all* states. Still, perhaps it is possible to learn an optimal policy at least at the *relevant* states: i.e., those which are encountered often. This idea of using policy iteration to try to learn optimal actions at relevant states dates back at least to TD-Gammon [6].

1. Initialize $\pi$ to a prior policy.
2. For $K$ iterations (i.e., learning epochs):
   (a) Use $\pi$ to walk the tree, generating sample data.
   (b) Use your favorite regression method to learn $V^\pi$ from the data.
   (c) Construct a new policy $\pi$ from the old policy $\pi$ and the value function $V^\pi$.

Table 1: Policy Iteration: Alternating Policy Evaluation with Policy Improvement

Recall that policy iteration alternates between policy evaluation and policy improvement. (See Table 1.) The key question we face when learning to play games with large state spaces is: how do we to walk the tree (Step 2(a)) to generate *relevant* sample data? That is, how do we use the information contained in the current value function—the only information we have—to seed these walks, to learn about relevant states?

[3] A point of clarification about our nomenclature: We use the term "heuristic evaluation function" when discussing a heuristic that is applied to a node/state to estimate its value (i.e., which player will win). We then use the term "value function" to denote the values are all nodes/states that have been backed up (in the sense of minimax or, equivalently Bellman, in our MDP formulation of the adversarial search tree) throughout the game tree/MDP.

A popular way to tackle this problem is via *Monte Carlo Tree Search* (MCTS), which uses repeated Monte Carlo sampling from the root to build a subtree of the (intractable) game tree. The data then comprise all the nodes in this subtree together with their estimated values. MCTS employs not one, but two, policies as it builds this subtree of nodes and their estimated values. It employs the *tree policy* if ever it encounters a node it has visited before (i.e., a node already in its subtree); and it employs a *rollout policy* if ever it encounters a node it has not visited before (i.e., a node *not* already in its subtree) in order to assess the value of that node, and then it adds it to its ever-growing subtree/data set.

1. Clean the slate: i.e., start from an empty data set.
2. For $M$ simulations:
   (a) While the game is not over:
      i. If the current node has been visited before:
         A. Make a move using the tree policy $\pi$
      ii. If the current node has *not* been visited before:
         A. Run $N$ rollouts to depth $d$, returning the average value of all nodes reached at depth $d$
         B. Optional, but very common: Walk back up the tree, averaging this value estimate for the current node into all its ancestors' value estimates
         C. Optional: Update the tree policy using the new information gleaned from the rollouts
3. Create a data set consisting of all nodes visited by the tree policy and their estimates. (Note that reliable estimates are only produced for nodes visited by the tree policy.)

Table 2: MCTS, for use in Step 2(a) of Policy Iteration

The simplest tree policy is an $\epsilon$-greedy policy, which chooses an optimal action w.r.t. the current value function with probability $1 - \epsilon$, and one of the other actions uniformly at random with total probability $\epsilon$. The simplest rollout policy generates only a single action (i.e., $N = d = 1$); that is, the most straightforward way to estimate a value at a state is to use its 1-step Bellman update (i.e., TD(0)), *à la* TD-Gammon [6].

A more sophisticated, and typical, rollout policy (which gives the policy its name) is one that simulates the game multiple times at the current state to (Monte Carlo) estimate that state's value. In other words, it runs *inner* simulations within the *outer* MCTS. The term *rollout* is used to describe one of these inner simulations. By generating multiple rollouts at a state, using a policy based on the current value function (e.g., $\epsilon$-greedy), and then averaging the rollouts' values, we produce a TD(1) estimate of the current state's value based on $d$ steps of lookahead, where $d$ is the depth of the rollout. After enough[4] learning epochs, it becomes natural to estimate the value at a state $d$ steps ahead of the current state using the current value function; but early in the learning process, it is natural to estimate this value using an heuristic evaluation function (e.g., the Voronoi heuristic), as was the practice[5] in AlphaGo [4]. Finally, the current state's new estimate can be averaged into the value estimates at all its ancestors in the tree.

An alternative tree policy that works in conjunction with a rollout policy would be an $\epsilon$-greedy policy again, but one based on the (new and improved) estimated values produced by the rollouts, instead of the current value function. More sophisticated still would be a tree policy that trades off between exploration and exploitation using multi-armed bandit technology: i.e., choose "arm" $j$ that maximizes

$$\bar{x}_j + c\sqrt{\frac{\ln n}{n_j}} \ ,$$

[4]Say $L$, another hyperparameter.

[5]It is common to speculate that the initial heuristic evaluation function, which was arrived at by learning from human expert games, was the secret sauce in AlphaGo [4]—essentially, seeding policy iteration with a smart prior policy. This hypothesis was debunked, however, when AlphaZero [5], succeeded without a smart initialization—from essentially a blank slate.

where $\bar{x}_j$ is the average value of arm $j$, $n_j$ is the number of times arm $j$ has been selected, $n$ is the number of times the current state has been visited, and $c$ is a constant, often $\sqrt{2}$. The exploitation side of this policy is guided by the first term, $\bar{x}_j$; arms with higher values are more likely to be selected. The exploration side is guided by the second term, which represents the width of the confidence interval associated with the estimate $\bar{x}_j$. Arms that have not been explored sufficiently have wider confidence intervals. This tree policy favors exploring these arms until their confidence intervals shrink.

In summary, most implementations of policy iteration run some variant of MCTS in Step 2(a), thereby simulating playing the game, making moves at each state based on the current value function, all the while collecting data (i.e., new and improved estimated values) at the states visited during these simulations. In the usual RL fashion, these new and improved estimates can be 1-step Bellman updates (i.e., TD(0)), or Monte Carlo estimates (i.e., TD(1)), or anything in between.

Adversarial search algorithms make the assumption that the opponent is a rational agent. RL for game-playing, in contrast, attempts to learn a best response to a given opponent strategy—rational or otherwise. This strategy is encoded in the transition probabilities of an MDP, and thus is invoked during the data-generation step in policy iteration. In particular, whenever a tree policy or a rollout policy is simulated, the next state in the MDP is a node two levels down in the adversarial game tree, which obtains first by following the learner's tree or rollout policy, and second by simulating the behavior of the opponent. In symmetric games like Tron, an alternative to learning given an opponent' strategy is *learning in self-play*. In this setup, the rollout policy still simulates both players actions, but it uses a recent[6] policy. As for the tree policy, after it takes its action, the learner can assume the role of the opponent. Learning in self-play can generate more robust strategies than learning a best response to a specific opponent's strategy.

# 5 Code

We have taken the liberty of implementing Tron-141 for you. We have implemented it as a derivative of `AdversarialSearchProblem`, from Assignment 2: i.e., `TronProblem` inherits from the abstract class `AdversarialSearchProblem`. Moreover, `TronState` inherits from `GameState`. Thus, your only task is to implement an AI bot to play the game.

## 5.1 Code to Modify

The code you should modify can all be found in two files:

- `bots.py` contains stencil code for the `StudentBot` class, where your main task is to complete the `decide` function, which indicates the move your `StudentBot` decides to take.

  This module also contains code for `RandBot` and `WallBot`, two bots against which you can test your `StudentBot`. You can also write your own baseline bots in this file to test your bot against.

  **Hint**: We recommend you read through the code for the bots we have already implemented. Doing so will help familiarize with the different members of `TronProblem` and `TronState`.

- `support.py` contains a function called `determine_bot_functions` to which you can add clauses that correspond to new bots you write in `bots.py`. This is only necessary if you create a bot other than `StudentBot` for the purpose of testing `StudentBot`.

**Note:** You may *not* use the `signal` library or catch `support.TimeoutException` as part of your solution.

## 5.2 Code *not* to Modify

**Note:** Do not modify any of these stencil files.

[6] For example, the most recent. It does not use the current policy to encourage stability, and ultimately, convergence.

- `tronproblem.py` defines the `TronProblem` and `TronState` classes. The function in this file that we expect to be most useful to you is the static method `get_safe_actions(board, loc)`, which returns the set of actions a player can take from the position `loc` that would not result in a collision.
- `gamerunner.py` runs the game. We describe some of its command-line arguments in the next section.
- `trontypes.py` contains constants that are used to identify cells on the board and types of powerups.
- `boardprinter.py` handles printing the board and game information to the terminal.
- `adversarialsearchproblem.py` is identical to the file of the same name we distributed with the Adversarial Search assignment. The `TronProblem` class inherits from the `AdversarialSearchProblem` class, and `TronState` inherits from `GameState`.

### 5.3 Testing your Solution

As you develop your bot, you can evaluate it by playing matches against various opponents on a variety of maps. Specifically, we are releasing four TA bots and eight sample maps. You can test your `StudentBot` in simulated games against other bots using the `main` function in `gamerunner.py`. This function takes a few command line arguments, the most important of which are:

- `-bots` lets you specify which bots to play against one another. The syntax is `-bots <bot1> <bot2>`
- `-map` lets you select the map that the game is to be played on. The syntax is `-map <path to map>`
- `-multi_test` lets you run the same game setup (choice of bots and map) multiple times. You may want to run multiple tests with the `-no_image` flag, so the games are played more quickly. (Printing to the terminal slows things down.) To do so, use `-multi_test <number of games> -no_image`.
- `-no_color` runs the game without coloring the board printout. You should use this option if coloring causes display issues.

For example, you can test your `StudentBot` against `RandBot` on the `joust` map using
`python gamerunner.py -bots student random -map maps/joust.txt`
You can test your `StudentBot` against `WallBot` 100 times with no visualizer on the `empty_room` map with
`python gamerunner.py -bots student wall -map maps/empty_room.txt -multi_test 100 -no_image`
**Note**: When running multiple tests, your bot will move first in every other match.

**Opponents** There are four sample TA bots:

1. `RandBot` chooses uniformly at random among all actions that do not immediately lead to a loss.
2. `WallBot` hugs walls and barriers to use space efficiently.
3. `TA-Bot1` and `TA-Bot2` Two more sophisticated TA bots, with secret implementations.

As already mentioned, you can find the code for `RandBot` and `WallBot` in `bots.py`. The implementation of the other TA bots is not exposed. Instead, it is included as a compiled module, `ta_bots.so`. You can still test your bot against these bots: when running `gamerunner.py`, use the `-bots` flag with `ta1` or `ta2` as an argument.

In addition to these bots, you should save versions of your own bot as you work to improve it. Earlier versions can serve as baselines against which you can test later versions, to be sure that your strategy is indeed improving.

**Maps** There are eight sample maps, available in the `maps` directory. Two are empty maps, one big (13x13) and one small (7x7). You should use this small map for testing purposes; and you should feel free to create and test your code on other perhaps even smaller maps as well. There are two other big maps without powerups, and four with powerups. As noted previously, you need not tailor your bot to powerups.

All maps are stored in `.txt` files, using the same characters that appear in the board printout. The only exception is the `?` character, which represents powerups in the files. When `gamerunner.py` reads in the map files, each `?` is replaced by one of the four powerups, chosen uniformly at random with replacement.

# 6 Writeup

You and your partner (if you have one) should hand in a final writeup by **Monday, December 14 at 11:59pm ET**. In short, this writeup should describe your Tron-141 bot.

This project is very open-ended. There are numerous approaches you might try, only a few of which you can be expected to get working within the allotted time frame. Your writeup should include:

- The back story: What did you try first? What worked? What didn't work? How did you eventually arrive at your bot's present design?
- A description of how your bot works, sufficiently detailed so that the reader could replicate your bot.
- A description of your bot's known shortcomings, including how you would attempt to ameliorate them with more time.

# 7 Warmup

For this project, you and your partner (if you have one) will be assigned a mentor TA to bounce ideas off of. To make sure you get started thinking about this project immediately, you are required to meet with this TA by **Friday, December 7th at 5:59pm ET**.

To prepare for this meeting, you should familiarize yourself with the rules of Tron-141, and with the support code. You should understand how to implement a basic bot (e.g., one that makes a random move), and you should also have some ideas about how you are going to build a more sophisticated strategy.

Also by **Friday, December 7th at 5:59pm ET**, you and your partner should turn in an implementation of a basic bot, along with a short report describing your plans for a more sophisticated strategy. The contents of this report should form the basis of your meeting with your mentor TA.

# 8 Tournament

We will be running a daily Tron-141 tournament, beginning on **Friday, December 7th at 11:59pm ET**, so you can see how your bot stacks up against other students' bots. To submit your bot to the tournament:

1. Copy your code from `/course/cs1410/Tron/` to `~/course/cs1410/TronTournament/`.
2. Add a custom bot name, by adding the following to your `StudentBot.__init__` function:

   ```
   self.BOT_NAME = "My custom bot name"
   ```

   Be creative!
3. Submit your tournament bot using `cs1410_handin TronTournament`.

Please submit only one bot per group! The tournament will grab the latest submissions every night around midnight, run a tournament, and then post the results online here. The tournament winner will be showered in praise, and might even be awarded a cash prize!

# 9 Grading

Your bot should be able to defeat `RandBot` virtually all of the time, `WallBot` virtually all of the time on most maps, and `TA-Bot1` and `TA-Bot2` most of the time—all on boards without powerups. We will test your bots on the `empty_room`, `center_block`, and `diagonal_blocks` maps, as well as on at least one secret map. If you indicate in your writeup that your strategy is designed to handle powerups, then the secret map(s) will include powerups. See `rubric.txt` for more details.

# 10 Capstone

These are the additional requirements for those taking this course as a capstone:

1. You must work independently.
2. Your bot *must* incorporate machine learning.
3. Your writeup must include an evaluation of your machine learning approaches, including experiments demonstrating which approaches worked and why. While you should of course include plots optimizing your hyperparameters, you should also compare different algorithmic approaches—for example, by comparing the strategies they learn on small maps, after their hyperparameters have been optimized.

# 11 A Note About TA Hours

As already mentioned, this project is very open-ended. There are many viable solutions, and it is not obvious *a priori* what will work and what will not. As such, you should not come to TA hours expecting definitive "Yes, this will work" or "No, that definitely won't work" kinds of answers. Instead, you should view TA hours for this project as an opportunity to talk through your ideas to get a second (or third) opinion. You can also ask the TAs to review past course material with you as necessary.

# 12 Install and Handin

To install, run `cs1410_install Tron` in `~/course/cs1410`.

To hand in your warm-up code, copy your project from `~/course/cs1410/Tron/` to `~/course/cs1410/TronWarmup/`. Then run `cs1410_handin Tron` in `~/course/cs1410/Tron`. In addition, please submit the written portion of the Warm-up via Gradescope.

To hand in your final code, run `cs1410_handin Tron` in `~/course/cs1410/Tron`. In addition, please submit the written portion of the final project via Gradescope.

**Please submit only one warm-up and final project per group!** When submitting on Gradescope, you must specify all members of your group. Note that you can do this after you upload your document and hit "Submit." Use the "View or edit group" option at the top right of the page.

Finally, please note that since this is a final project, the normal resubmission policy does not apply. You may not use any late days. **December 14 is the hard deadline for all parts of the project.** Do not wait until the last minute to submit. Submit on Sunday, and update on Monday as necessary.

In accordance with the course grading policy, your written homework should not contain your name, Banner ID, CS login, or any other personally identifiable information.

# References

[1] N. Den Teuling and M.H.M. Winands. Monte-carlo tree search for the simultaneous move game tron. In *European Conference on Artificial Intelligence Computer Games Workshop*, September 2012.

[2] M. Lanctot, C. Wittlinger, M. Winands, and N. Den Teuling. Monte carlo tree search for simultaneous move games: A case study in the game of tron. In *Twenty-Fifth Benelux Conference on Artificial Intelligence*, November 2013.

[3] S. Samothrakis, D. Robles, and S. M. Lucas. A uct agent for tron: Initial investigations. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 365–371, 2010.

[4] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science*, 362:1140–1144, 12 2018.

[6] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

# A Powerups

Below are two example 13x13 game boards *with powerups*. The one on the left is the initial board, and the one on the right is the same board after Player 1 has moved down and Player 2 has moved up.

```
###############
#1            #
#             #
#       *     #
#             #
#    @     ^  #
#             #
#        !    #
#           2#
###############
```

```
###############
#x            #
#1            #
#       *     #
#             #
#    @     ^  #
#             #
#        !   2#
#           x#
###############
```

The `*`, `@`, `^`, and `!` symbols represent powerups (traps, armor, speed, and bombs, respectively), which players obtain by moving into a cell that contains one. Powerups are not an essential aspect of Tron; they were invented by past CSCI 1410 TAs. A player obtains a powerup moving into a cell that contains one. There are four different types of powerups.

**Trap** powerups create up to three new barriers on the border of the 5x5 area surrounding the opponent. The -'s on the board below denote the locations at which barriers could be placed near Player 1, if Player 2 moves into a cell with a trap powerup. The new barriers' locations are selected uniformly at random among the unoccupied cells on this square. (If fewer than three cells on this square are unoccuppied, fewer than three new barriers are created.)

```
###############
#             #
#    -----    #
#    -   -    #
#    - 1 -    #
#    -   -    #
```

```
#    -----    #
#             #
#           2 #
###############
```

**Armor** powerups allow a player to travel through a single barrier. After a player obtains an armor powerup, it is applied automatically, if ever the player moves into a cell with a barrier. Note that an armor powerup only allows players to travel through *barriers* (represented on the map by `x`'s), not through permanent walls (`#`) or other players.

**Speed** powerups are like a speed boost. They afford a player four consecutive, *mandatory* moves.

**Bomb** powerups destroy all the barriers (`x`'s) in the 9x9 area surrounding the bomb, replacing them with open space. They are activated immediately when a player moves into a cell that contains one. The `-`'s on the board below denote the locations where barriers would be destroyed if the bomb in the center exploded.

```
#################
#               #
#               #
#   ---------   #
#   ---------   #
#   ---------   #
#   ---------   #
#   ----!----   #
#   ---------   #
#   ---------   #
#   ---------   #
#   ---------   #
#               #
#               #
#################
```