COS4070 - Project 1

Foxes Drink Beer; Hounds Jail Foxes

Version 1.2

Z. Hutchinson
zhutchinson@aubg.edu

September 2, 2025

1 Due Date

A final submission is due: **Oct 12th by midnight**.

This is a hard deadline. Any submissions after midnight will receive a 0.

2 Project Files

You can find the project 1 code in one of two places depending on how you like to build C++ projects:

For those of you using Visual Studio, you should clone the following github repo:

https://github.com/zaxhutchinson/Microworld1_VS.git.

This is primarily geared toward Windows users.

For those on Linux or Mac, you should clone the following repo that uses a simple makefile:

https://github.com/zaxhutchinson/Microworld1.git

3 Goals

The primary goal of this project is to give you the experience of working on a modestly complex AI task. This goal is achieved if you:

- Feel overwhelmed by the many possible, albeit imperfect, solutions.
- Make peace with failure.
- Feel happy when ideas work some of the time.
- Get excited when something unexpected happens.
- Realize programming common sense knowledge is very hard.
- Get to a point where you have no idea how to improve your solution.

Therefore, do not be dismayed by failure. Keep trying. Be prepared to start over several times in whole or part.

AI, like biological intelligence, is created by a journey with many ups and downs. It is not created in a day, born perfect in every way.

4 General Instructions

Read this document fully and carefully. The devil is in the details. If something is unclear either ask in class or stop by my office.

Your task is to conceive of and engineer two different AIs. One AI will control all foxes. One AI will control all hounds. The AIs will share some similarities, but as their tasks are different, they should also have some substantial differences.

If you find any bugs that either crash the program or deviate from the description below, please email me right away. I will fix the bug and upload a new version.

5 Background

In this simulated work, there are foxes and hounds. Foxes are hedonists who only like to drink beer. They do not contribute to society whatsoever. The hounds, who love order and progress, take offense at the foxes behavior. So to teach the foxes a lesson, the hounds spend all their time running around trying to arrest the foxes before they can find and escape with all the beer. As you might guess, it is a rather unproductive society.

You, as the AI programmer, will be playing both sides. You will write AI routines to control both the foxes and the hounds. Each one has different motivations and skills which are addressed below.

6 The Importance of Limited Information

Your solution **may not** rely on modifications to any of the other files included with the simulator. If you add files to the simulator to break up your AI code, you may not include simulator files in these additional files except what is included in the ai.hpp file. Your AI files may not read in the world text file or any other files included in the simulator.

Reasons for the above: Each type of AI (Fox and Hound) are given different information and capabilities. Their views of the world are limited in different ways. This is intentional. Your AI must operate using only the information provided by the simulator either during object instantiation or in the call to *RunAI*. No additional simulator information is allowed and will be considered cheating.

7 Environment

The environment is a 2D grid-based map consisting primarily of walls and open spaces. Additionally, open spaces can contain one of several special properties. These are all described below.

The environment is generated from a text file found with the simulator. Different characters in the file specify different types of map cells.

List of cell characters:

- 'w' is a wall. Walls are the only impassable cell type.
- 'o' is an open space without any special properties.
- '@' is a starting location for foxes.
- '$' is a starting location for hounds.
- ']' '[' '(' ')' are all teleporter cells.
- '!' is a bottle of beer, i.e., goal cells.
- '?' are exit cells.

Several test environments are provided with the simulator. You are free to make your own to further test your AI (this is encouraged). If you do, please note that all environments must contain wall cells along the exterior of the map.

7.1 Special Cells

Teleporters

Teleporters are cells that can only be seen by and used by foxes. If a fox uses a teleporter, it will be instantly moved to another location on the map. There are four possible teleporter symbols: [ ] ( ). Not all of them have to appear on the map, but there will be no more than four. They can be linked in any way. This means that using a [teleporter might take you to the) teleporter. However, using the) teleporter might take you to the (teleporter. A teleporter can even link to itself, but I will not give you scenarios where this happens. They are given distinct symbols so that your AIs can learn which symbols link to which other symbols. Again, hounds cannot see or use teleporters. Think of these as perhaps holes or gaps in the hedgerows or rocks through which the foxes can slip but the hounds cannot follow. Teleporters can be used repeatedly.

Teleporters make the environment more difficult because using a teleporter might take the animal to a region of the map it could not otherwise reach. But it could also take the animal to a part of the map it could reach without the use of the teleporter. This creates cycles in the environment which, unless properly remembered, could cause an animal to go in circles.

Goals

Goal cells (or bottle of beer) are how foxes score points. Each beer is worth 1 point. When a goal location is used, it disappears from the map and becomes a regular open cell.

Exits

Exits cells are designated by question marks (?). They are how foxes leave the map and enjoy their drinks. Once a fox exits the map it is finished and cannot reenter the map. However, it also cannot be captured by the hounds. All points it has scored prior to exiting are secure.

Exits create tension for foxes. If they exit, they keep their points. However, if they exit, they cannot score more. Foxes should exit if they feel in danger but not frivolously.

Starting Locations

Starting locations are used only to place foxes and hounds at predefined locations on the map. Once the simulation starts, starting locations are invisible to both parties. They also cannot be seen in the display. NOTE: if there are more foxes/hounds than starting locations, some will start in the same cell.

7.2 Percepts

The simulation runs turn-by-turn. Each turn all foxes move first followed by all hounds. Each animal is provided with a *percepts* object (see *percepts.hpp* file for details). This object contains what the animal can see or smell. The percept object contains seven data points. It contains five vectors of single character strings corresponding to cell contents. There is a vector for what is in front of the animal, behind, left and right. There is also a vector for the cell underneath the animal; however, this vector always has only one string.

The contents of each of the four directional percept vectors (forward, backward, left and right) is limited by one of two things: walls and sight limitations. Vision extends until a wall is encountered (sight cannot pass through walls) or until the animal's max sight is reached.

Within the percepts object is a sixth vector of sighted animals (a vector of Sighting objects). If one or more animals is sighted, this vector will contain the type of animal (AgentType enum), the direction as a single character string (F, B, L or R), and the distance. In other words, information about other animals is separate from the map information. All animals can see all types of animals. This means foxes can see other foxes as well as hounds (and the same for hounds).

The seventh data member of the percepts object is a double representing an animal's ability to "smell" other animals. Scent contains the straight-line distance to the nearest animal of the opposite type. Be wary of using this value only to guide an animal. While sight does not pass through walls, scent does. It is possible that "following one's nose" will not result in finding the other.

7.3 Fox Percepts

Foxes see everything allowable (teleporters, exits, goals, walls, open cells and other animals). Foxes also see in three directions. All animals can see what is in the cell with them.

Foxes do not receive any useful scent information. They only ever receive a scent value of 0 no matter how close or far a hound is from their position.

7.4 Hound Percepts

Hounds see very little. They only see walls, open spaces and other animals.

Hounds *do* receive useful scent information. The value in their scent percept contains the straight-line distance to the nearest fox. Again, be wary that scent passes through walls. And also, that on subsequent turns, the nearest fox might change although the scent value is constant.

8 Commands

Each turn, each animal's RunAI must return a vector of single character strings. What is in this vector is the task of your AIs. The vector can be empty, in which case, the animal does nothing. There are five possible commands:

- F: move forward
- B: move backward
- L: turn left
- R: turn right
- U: use current cell

9 Movement and Turning

Each animal has a facing within the environment. The animal will never know precisely which way is north or south. It does not know its own absolute orientation. However, the simulation keeps track of the animal's absolute heading. If an animal is facing south and moves forward, it will move one cell to the south. If it turns left, it will now be facing east.

For example, if you wanted the animal to turn right, your vector would contain a single "R".

10 Animal Actions and Differences

10.1 Movement

Animal types have different speeds. Speed indicates how many commands can be returned from the RunAI function. Each one will be executed in the order they are added to the vector. For example, were an animal with three speed to return a vector containing "L", "F" and "U" the simulation would cause it to turn left, move forward and then use the cell (provided the animal type could do these things).

If an animal returns more commands than its speed allows, the additional commands are ignored. An animal can return fewer commands than its speed allows.

Foxes are slower than hounds and can move two cells each turn (they're always a little tipsy).

Hounds can turn left or right, but can only move forward.

Foxes can turn left or right and can move both forward and backward.

Grading scenarios will use the following values:

1. Hound speed: 3

2. Fox speed: 2
3. Hound sight
4. Forward: 4
5. Left and Right: 1
6. Backward: 0
7. Fox sight
8. Forward, Left and Right: 999 (effectively infinite).
9. Backward: 0
10. Hound actions:
11. F, L, and R
12. Fox actions:
13. F, L, R, B, and U

11 Using Cells "U"

Special cell properties **are not** invoked on simply moving into the cell. They must be used by the AI adding a "U" to the returned commands vector.

Foxes can use all the cells.

Hounds cannot use cells. A hound AI emitting a "U" will do nothing.

Using a teleporter teleports the animal to the linked teleporter.

Using a goal cell grants the fox a single point and causes the goal to disappear. NOTE: moving over goals does not score points. The AI must recognize when it is standing on a goal and "U"se it.

Using an exit removes the animal from the map.

12 Agent Communication

Hounds have the ability to communicate with each other. **Foxes do not.** The RunAI method in the AI, FoxAI and HoundAI classes take a second parameter, a pointer to an *AgentComm* struct. You can find the declaration of the AgentComm struct in the *comm.hpp* file.

An AgentComm struct contains two data elements: bark and direction.

- direction: a vector of Vec2 objects. This vector will contain a relative directional vector to all hounds (including the current one).
- bark: a vector of unsigned integers each hound can use to send messages to the other hounds.

There are several things to note. These vectors are keyed to the hound's id (because ids are assigned starting with 0). Therefore you can use a hound's id to index both of these vectors to read (or in the case of the bark, store) information.

The direction vector is created for each hound and takes into account the hound's heading. There is no point changing the data in this vector because it will be emptied and recreated for the next hound.

The bark vector, however, is never emptied by the simulation. It persists across all AI calls. If a hound does not change its own "bark", then it will never change. The idea behind the bark vector is that it allows hounds to talk to each other in a very simplistic way: one unsigned integer. Hounds are not very smart and their brains can only hold the equivalent of a single integer. It is up to you to encode messages in this integer. Of course, I say, "simplistic", but even a 32-bit integer allows the hounds to say approximately four billion different things. A 64-bit integer is, for human purposes, infinity.

As I stated above, foxes cannot communicate. Therefore, the FoxAI's RunAI method will receive a *nullptr* as argument to this parameter. Foxes are self-absorbed, narcissistic drunks who do not care what other foxes are doing or suffering.

The hounds will receive a pointer to an AgentComm struct. The vectors within the AgentComm struct will have as many data elements as there are hounds.

For example, a hound with id 0 could use the AgentComm bark at index 0 as its place for messages to the other hounds.

The three int data members of AgentComm can be used however you like to communicate between hounds. For instance they could encode an alert to the other hounds that they've seen a fox. You will need to devise an encoding. You may make it as complex as you wish, but you only have three integers to play with...because hounds are not very smart.

13 Objectives

13.1 Foxes

Foxes score points by finding and using goal cells. Each goal used gives the fox 1 point.

A fox must balance the need to explore to find goals but also to exit so as not to be caught. If a fox is caught, any points it scored are given to the hounds (and the fox gets nothing).

13.2 Hounds

Hounds score points by arresting foxes. They do this by simply moving into the cell with a fox (or if a fox moves into a cell with a hound). When this happens, the fox is removed from the map and the hound is given the fox's points (if it has any).

Any goals left on the map are awarded to the hounds.

13.3 Goals

The number of goals will not be known by either side.

14 Starting Locations

You can code your AI routines assuming the following are true:

1. All hounds will ALWAYS start in the same location.
2. No guarantee with respect to starting location is made for foxes. On some maps, they might start in the same place. On other maps, they might not.

15 The End

There are two ways the simulation ends. 1) If all foxes have either been caught or have exited the map, the game ends. 2) If the max turn limit is reached, the game ends.

16 Collaboration

You may work in groups of *two* if you so choose. Not three. Not four. *Two*. Don't ask for more. **This is not a requirement.** This might sound beneficial but there are several important points to consider.

- **You must include in your write-up who did what. And I want details. You must also state in the write-up the names of both people. You must each submit the EXACT same files: code and write-up. If your files differ, you will lose points.**

- I will expect two-student submissions to exhibit approximately double the effort of single-student submissions. This does not only mean more code. This includes the sophistication and breadth of the solution and the same reflected in the code, the details in the write-up...everything.
- I will not arbitrate group issues. If you start your project working with someone else and you find you cannot work with them. Or that they are holding you back. Or they drop the course. This is your problem to solve. I suggest you agree beforehand to several points (these are just suggestions):
  - that if you part ways, the moment you part ways, both parties share the code as it is.
  - both parties always keep a local copy of any changes.
  - you agree that the partnership can be dissolved by either party at any time.
- As part of the previous point, it is your responsibility to divide the work evenly. I will not listen to arguments that a failed project is due to the other person and that *your* contribution deserves to pass.
- If a group splits, the two members cannot form new groups with other students. They can reform the same group. But they are not allowed to form new groups with other students.
- There is a cliche that says, "Two heads are better than one." But in software development, we have a saying (from the great Fred Brooks), "Adding more manpower to a late project makes it later." Unlike, say, digging a ditch, more developers working on one piece of software does not necessarily speed up development or make it better.
- Friends do not let friends freeload. If you find you are doing all the work, politely go your own way. Team work is not a license for one half to carry the other.
- These are a risks. Take the partnership seriously. You are adults.

If there are so many downsides to team work, why do it, you might be thinking. Because team work is the way the world works. Team work is the way great ideas are born. There is so much to be gained by working through a problem with someone else. It can be so rewarding when it goes right. I want you to have the opportunity, if you want it.

Just be wise about it. As with everything.

17 Grading

Below are several points concerning the grading including the point breakdown.

17.1 Requirements

- **Grading will be done using an environment (i.e. map) not included with the project files.** This means your AI implementation cannot "hard-code" environment specific solutions.
- Your AI routines can only use simulation and environment data provided by the RunAI function parameters. Solutions that attempt to circumvent this will be given a 0.
- The map symbols defined in the config.json file will not change during grading. In other words, your AIs will not need to deduce which symbols have which properties.
  - A partial exception to the above: teleporter destinations will change. But the symbols for teleporters will not.
- If your solution contains debug or logger output, please disable this output for your submission. I do not want to run your code and see thousands of lines of output to the terminal or a file.
- Make sure your code is readable. If comments are necessary to maintain readability, add them. If something goes wrong (crash), I will try to locate the source of the crash and the reason for it. If it's something very minor, it will cost you only a few points. If it is major, it will be more. However, if I cannot decipher your code in a reasonable (read, short) amount of time, the bug will be considered major. Therefore, readability is in your interest.

- Your write-up should contain the following:
  - A separate English description of your solution for both the fox and the hound. It should be written in such a way that a non-programmer can understand it. We will have a homework early on in which you describe initial ideas before you start to code. I want to hear which of those worked (if any) and which did not.
  - Describe any behavioral problems you have identified with your solutions. Behavioral is a high-level description of how the animal reacts in a given situation. You don't need to describe all problems. Just the worst one or two. They can cover one or both animal AIs.
  - Lastly, I want a section in your write-up on your personal experience creating a solution for this project. It does not have to be long but it must contain something meaningful.

17.2 Grade Breakdown

Grades are out of 10 points (5 points per AI). Here are a list of issues that can cause you to lose points. This list is not exhaustive. This list does not have points attached. The number of points will be assigned according to the severity of the issue. In general, you need to make a strong effort to implement two AIs that solve the two problems and convince me of your work. You should be able to verbalize your approach. Lazy solutions will not score well.

- Crash (program ending bug).
- Incomplete or unfinished AI (fox or hound).
- AI Implementation (fox or hound) does not attempt to solve the problem (purely random solutions, phony solutions, obviously bad solutions, etc.).
- Hounds do not make good use of the AgentComm data (they don't talk to each other).
- AI does not map the environment.
- AI does not make use of all percepts.
- AI does not return the full range of commands or they are returned randomly or in the wrong context.
- Code readability.
- Missing, lazy or inadequate write-up.

18 Submission Instructions

You must submit via Canvas your modified *fox_ai.hpp*, *fox_ai.cpp*, *hound_ai.hpp*, *hound_ai.cpp* files and a write-up as a **PDF** or plain .txt file. These are the only two acceptable write-up file formats. If your solution utilized additional code files, you must submit those, too.

When grading I will substitute your files for the four ai files (and make available additional AI files you upload). If you submit simulator files, modified or not, they will not be used and points will be deducted.

19 Installation and Running

19.1 Installation

This project requires that you have SFML-3.0.0 installed. Here is a link to instructions if you are using Windows and Visual Studio. SFML Installation Instructions On this site, there are also instructions for Linux, CodeBlocks, and Mac. I will go over installation in class. If you miss it or are having trouble, **please stop by my office!**

I am providing both a Visual Studio project and a zip containing a simple makefile. If you are working on Linux or Mac, try the makefile version. The makefile version was tested on Linux (Manjaro) and it might need a little reconfiguring for Mac or Windows.

19.2 Running

Microworld: Foxes and Hounds supports a number of command line arguments.

- -t <int> The -t argument followed by an integer specifies the max turns a simulation will run. If max turns are reached, the simulation ends and current points are displayed.
- -d <int> The -d argument followed by an integer specifies in milliseconds the time between AI updates. For example, -d 1000 gives 1 second between turns. Lower values will speed up the simulation.
- -w <string> The -w argument allows you to specify different world files to use. For example, -w world1 would load the simulation from a file *world1*. If -w is omitted a file called *default_world* is looked for.
- -rs The -rs argument enables random start positions. Use of this argument only makes sense if there are more starting positions for an animal than there are animals in the simulation.
- -rh The -rh argument enables random headings for all animals. If -rh is omitted, all animals begin the simulation facing north.
- -rsh (or -rhs) This argument (and its variant) enable both randomized starting locations and randomized headings. **This argument will be used for grading.**
- -s <unsigned int> The -s argument seeds the simulator's random number generator with the unsigned int provided. This allows you to use the random arguments above but replay runs with deterministic behavior. The random number generator is also passed by pointer into each type of AI and stored as a private member variable. If you base any randomness within your AI routines on this rng variable, they will also be deterministic when the seed is set. Can be handy for testing. Omitting this argument causes the simulator to use std::random_device to seed the rng.

19.3 Hotkeys

Escape will immediately terminate the simulation without displaying the final score.

P will pause the sim (if you want to study some aspect of it). Note, that is does not pause the inter-turn delay. If you have, for example, a 5 second turn delay and you pause it for more than 5 seconds, as soon as you hit *P* again, the next turn will immediately run.

20 Advice

Start small. First, use pencil and paper to plan the AI for each animal type. Think of the different micro scenarios that might happen. What should the animal do in that situation?

Next, how will animals explore the map? How will they using pathfinding to go from one known location to another? How will they store the map as they explore it? Work on support code (functions) that perform common tasks.

Consider using concepts like belief and uncertainty to guide animals in meta ways. If they are mapping the environment and keeping track of what they've seen, can they logically conclude certain things? Where is it safe? Should it relocate? Is it in danger of being trapped? Finding ways to (sometimes) utilize logically acquired meta knowledge is what separates a reactive, simple solution from a thoughtful, complex one.

Use simple maps to test your solutions. Rearrange the map to set up different scenarios. Then move to larger maps. Test, test, test.

While pure random solutions are not acceptable, your solution should use randomness in some way (of course, it does not have to). Random solutions allow agents to perform more toward the average on any given map. Deterministic solutions can are very dependent on a) the map and b) where they start.

Remember percepts are collected *animal-by-animal* rather than once for all each turn. What does this mean? This means that when it is an animal's turn to act (when the AI code runs), what is sees is the actual state of the world.[1]

Have fun. AI is about trying to solve hard problems with risky (e.g, high probability of not working), avant-garde ideas.

[1]This is in contrast to turn-by-turn where all information is collected at the start of the turn for all animals and all actions are executed at the end of a turn. Depending on an animal's order within the turn, the world might have changed.