Problem.

You are going to reproduce an experiment first conducted at Harvard University in 1968. This assignment is a bit different from the other homework assignments in this class because there are no specification bundles for this. You either make it work or you don't. You are going to study the effects of evolution on a population of robots. The robots need to maneuver around a grid collecting energy. The robots must collect more energy than they expend to survive. Your robots will maneuver around a 10 x 10 grid looking for batteries. A battery gives the robot five more power. Moving a square costs 1 power. The sensors are always on and cost no power. Robots have five power when they first emerge on the map.

The key is the robot behavior. Each robot has a collection of direction sensors and a motor. Robot success depends upon the map between these two elements. We do not hard code the map between the sensor data and the motor actions, however. The robots start with a random mapping, but over time, the mappings evolve into successful strategies. We'll get to how they do this in a minute.

Robot Sensors.

The robot has four sensors - each facing in a different direction. This way, the robot can detect what is in the squares adjacent to the North, South, East, and West. Each map feature will generate a different code for that sensor. See figure 1 for four examples. The codes you see here are examples, you can use whatever codes you wish.

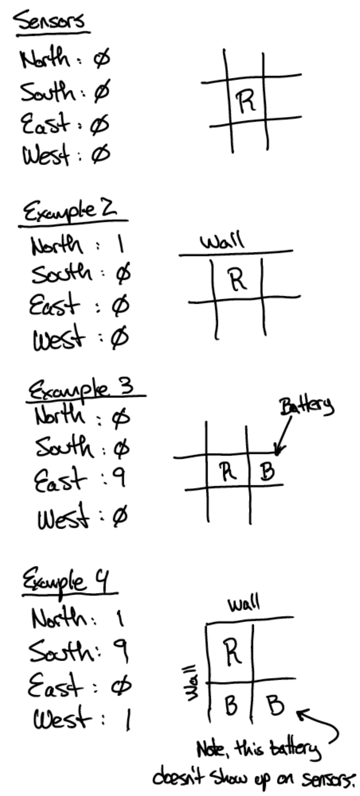

Figure 1: Example of sensor readings with various obstacles on the map.

Robot sensor states:

- No object in square.
- Wall object.
- Battery object.
- Don't care if anything is there.

Robot Genes.

Each robot will have 16 genes. Each gene is an array containing five codes. See figure 2 of a single gene. The first four codes correspond to possible sensor states. The last code instructs the robot what to do in the event the current sensor state matches the four states on the gene.

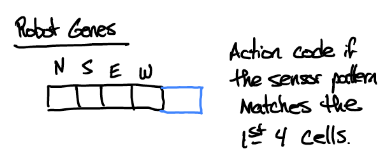

Figure 2: Example of an individual robot gene.

Robot actions:

- Move 1 north.
- Move 1 south.
- Move 1 east.
- Move 1 west.
- Move 1 random direction.

This means that every turn the robot is comparing the current sensor state with it's genetic code in an attempt to find a match. A gene must match exactly for it's behavior to be executed. If it does not find an exact match, the robot will execute the last gene (number 16). Robots will continue to do this until they run out of energy. KEEP TRACK OF THE NUMBER OF TURNS A ROBOT SURVIVES. This will become very important later. A robot gains energy running into a square containing a battery. The battery is consumed in the process. Figure 3 is an example of 1 robot turn.

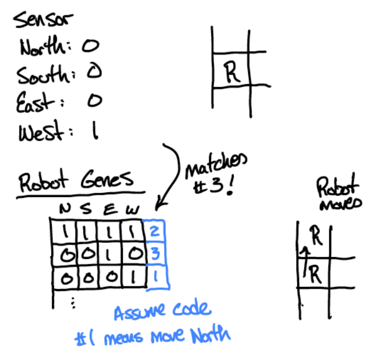

Figure 3: Example of a robot moving along the map for 1 turn.

The Map

Randomly place each robot on a spot on the map. Populate 40% of the squares with batteries. Generate a new map for each robot. We run one robot through at a time. Moving each square consumes 1 energy unit - even if it's not a valid move (a wall). Invalid moves consume energy, but keep the robot on the original square. This is usually the death knell for that robot unless the move action was move in a random direction.

You don't need to display the map on the console. It's interesting to watch if you want to do so, however. Perhaps display the map of the best performing robot or the worst performing one.

Robot Population

Evolution works on populations, not individuals. You need a population of robots to run through your maps. Create a population of 200 randomly generated robots to start. That is, you randomly generate the mappings between allowable behaviors and sensor codes for the first generate of robots only. Each robot is run through a random map and the number of turns they survive is recorded.

Robot Reproduction

Once all the robots in a population have had a turn (and acquired energy scores), record the total energy harvested by the entire generation and breed your robots. Sort your population by energy harvested and kill off the lowest 50%. Then pair up the top 2 robots and produce 2 children by combining genes from both parents. The children enter the gene pool with the parents in the next round. Then, breed the 3rd and 4th highest scoring robots. Repeat until all the parent robots have reproduced. Keep the number of robots at a fixed number for the entire simulation. Genes are randomly created for the first population only - the robots breed after that.

Each parent supplies 50% of the 16 genes for a child. The simplest way is for one parent to supply the first 8 and the other to supply the last 8. You will not want the siblings to have the same genes (unless you are investigating the effects of identical twins). However, you can use a different swap scheme if you would like.

Swapping genes is a tricky business and mistakes happen. In 5% of the individual genes swapping there is an error - a mutation. Randomly change one character on the gene the child has inherited from it's parent. Just generate another value and insert that value over the old one. You can also shift all genes down 1 and the 16th gene moves to the top.

Genetic Algorithms.

The degree to which the robot successfully harvests energy from the environment is called 'fitness'. We measure that by the total amount of power harvested when each individual robot's time ends. When we finish with the entire population of 200, we calculate an average fitness score for the population. Save the average fitness for each generation. You will most likely see slow and steady improvement over time - evolution at work. When the simulation completes, print out the average fitness scores on the console. This is even more effective if you are able to draw a console graph (not a requirement).

The fascinating thing about this experiment is your robots will get better and better at navigating the map over time. They will evolve strategies dealing with walls and corners. The most exciting part is it doesn't require any further participation from you - just set it in motion and watch it go.

Bonus Features.

No bonus features are needed with this assignment. However, past students have:

- Added obstacles for the robots to avoid.
- Added predator robots.
- Added vision so the robots can 'see' 2 spaces away.
- Added memory of moves.
- Plotted fitness over time on the console (a common modification).

- Created Don Juan robots that have children with several partners.
- Stored many of the constraints as constants so they could explore the effects of modifying them (ex: change the mutation rate to 8% from 5%) - another common modification.
- Keep track of how many generations a specific robot survived.

Homework Checklist

Check the following before you turn in your work:

- ☐ You coded your homework.
- ☐ Does it meet all the requirements?
- ☐ Test your code.
  - ☐ Does it compile?
  - ☐ Does it have any compiler warnings?
  - ☐ Does it run?
  - ☐ Does it produce correct output?
  - ☐ Did you use the grep trick to make sure I can see your work?
- ☐ Upload to Canvas.
- ☐ What's the plagiarism checker score?

Due Date

This assignment is due by the close of Canvas for the semester.

Late Work

There is no late penalty for this assignment because it is due when Canvas closes. You simply cannot turn it in after that.

Turn homework in by uploading to the appropriate Canvas Dropbox folder. Change the file extension from a .cpp file to a .txt file. This is because Turnitin does not work with .cpp files. Don't zip or otherwise compress your files.

I ONLY accept homework through the Canvas Dropbox. Do not add it to the dropbox comments or email me - I will not accept it. If you are having trouble submitting the assignment, email me immediately. Make sure you upload it a few minutes before the assignment closes in Canvas. If you go over by just one second - you are late.

Style Guide.

All programs you write MUST conform to the following style specifications.

Comments.

Use white space and comments to make your code more readable. I run a program called cloc (count lines of code) which actually looks for this stuff.

End of line comments are only permitted with variable declarations. Full line comments are used everywhere else.

Comment Rate.

I use a program called cloc to calculate the total number of blank lines, total number of comments and the total number of lines in your program. When you divide the number of comments by the number of lines you get the comment rate. This number gives me a rough approximation of how well commented your code is. When I see a rate of 10% (for example) I have a good idea of what I will find when I look at the program - pretty much no comments at all. Ditto, when I see a rate of 45% (for example). This code will have comments for many tricky or complex sections as well as functions.

Compiler Warnings.

Compiler warnings are a potential problem. They are not tolerated in the production environment. In CISP 360 you can have them. I will deduct a small number of points. CISP 400 - I will deduct lots of points if compiler warnings appear. Make sure you compile with -Wall option. This is how you spot them.

C++ Libraries.

We are coding in C++, not C. Therefore, you must use the C++ libraries. The only time you can use the C libraries is if they haven't been ported to C++ (very, very rare).

Functions.

Functions are used to segment your code into easier to work with chunks. You want your functions to do only one thing - one activity. Functions are coded BELOW main(), not above it. Use arguments and parameters to pass information to your functions - global variables are discouraged with prejudice . Whenever possible, try to have a brief comment below the function signature describing what the function does.

```
void foo()
// a meaningless function to show a function comment in
the style guide.
```

Function Prototypes.

Use function prototypes and comment them. This is a constraint in this class even if not expressly stated in the homework. I use the grep trick for Function Prototype to look for this. You only need to comment it once, at the top of your source file - above main(). Example:

```
// Function Prototype
void ProgramGreeting();
```

Non-Standard Language Extensions.

Some compilers support unapproved extensions to the C++ syntax. These extensions are **unacceptable.** Unsupported extensions are compiler specific and non-portable. Do not use them in your programs.

Program Greeting.

Display a program greeting as soon as the program runs. This is a description of what the program does. Example:

```
// Function Greeting
cout « "Simple program description text here." « endl;
```

You can make this much more elaborate - usually used as cover so clients don't realize we are loading huge data files. If your assignment calls for a menu, DO NOT put the menu in here - that goes in it's own section.

Source File Header.

Start your source file with a program header. This includes the program name, your name, date and this class. I use the grep trick for .cpp (see below) to look for this. I focus on that homework name and display the next 3 lines. Example:

```
// homeworkname.cpp
// Pat Jones, CISP 413
// 12/34/56
```

Space Rate.

I use a program called cloc to calculate the total number of blank lines, total number of comments and the total number of lines in your program. When you divide the number of blank lines by the total number of lines you get the space rate. This number gives me a rough approximation of how well laid out your code is. When I see a rate of 10% (for example) I have a good idea of what I will find when I look at the program - wall of words. Ditto, when I see a rate of 45% (for example). This code will have good spacing between major blocks of code and functions. Note: when this number get's too high (like 70% of so) the blank space becomes a distraction.

Variables.

Document constants with ALLCAPS variables and the const keyword. Magic numbers are generally frowned upon.

Grep Trick.

Always run your code immediately before your turn it in. I can't tell you how many times students make 'one small change' and turn in broken code. It kills me whenever I see this. Don't kill me.