PROFESSOR: So today's lecture is supposed to be about navigation, but in many ways, that can be a pretty dry subject. So what I've done is I've made it very graphical and sort of active and animated. No, this is just a folder. This isn't the-- Anyway, so a couple of administration pieces is the Sprint submission deadline. The Sprint Tournament will be held on Wednesday, but you need to submit your code by Monday at midnight. There's a little bit of wiggle room where you can submit it like a few hours after midnight, but don't plan on that. Yeah. And you submit your code to the Sprint Tournament and that makes you eligible to win prizes. And it also helps you with your seeding, so that you'll be seeded into a higher part of the tournament bracket for the next tournament. So yeah, you've got to get started early.

We'll also be offering another source of money for you, as if you didn't have enough sources of money with the Sprint Tournament, and all of those special prizes we're giving out, and the tournament itself. We're also offering an extra prize for the team that is most helpful to other teams and people. So that means that now you have an incentive, beyond just being a good person, to help other teams out. We'll have a survey that goes out in the last week. And so if you've gangstered everybody into voting you as helpful, then you'll win the prize without being helpful. So just keep that in mind.

All right. So today we're supposed to talk about navigation. Let's have a look at this thing. Here is a program that I wrote that makes a maze, piece by piece. There's probably a smarter way to do it, but I just built random walks that self avoid onto one another to make a maze.

And what I've done here is I've got a variable called delay that lets me change the speed at which a computation is performed. So you can see here, if I set it to 0.5,

then we'll be here all day, because each one of these paths is going to take a half a second to put together. But this is helpful, because a lot of times a pathing routine will happen instantly. Or in compute time, it'll happen fast, and you won't really notice what's going on.

So let's go ahead and set this to a small number. Now we've got a map that has a bottom left and an upper right. And we're going to use this map to demonstrate a bunch of pathing techniques. So we're trying to get from the bottom left to the upper right.

The first method we're going to use is depth first search. So I'm going to run this. Oh, it finished. That was dumb. Let's make the delay 0.1. Yeah. So now, you can see I'm starting from the left and I'm going along in a depth first search, which means that I'm looking toward the end first. And then after I have investigated everything, I'll start again at the next branch point. So I haven't even looked at this.

I've gone along and, yeah, so I check that-- oh, done. OK. So now you can see it finished. But I'm going to run it once more, and I want you to keep your eye on the colors that are displayed on the right here. The color indicates how far it thinks something is. And it's scaled from white to black. So when it finds something that's darker, it will brighten up this other area.

So pathing systems are basically just a way of trying all the possible combinations. But there are little tricks of making that smarter. I'll show it one last time. The point is to make it smarter than trying every possible combination.

Like for example, if you've visited an area before, then don't try that again. Or if you visited an area before, and you got there closer than you currently are, then don't try that again. So there you have it. There is an example of depth first search.

And we're going to move on and show an example of breadth first search to show how different it is. They're going to look pretty similar for this example, but the next example I give won't be pathing in a maze. It'll be pathing in something that's more open.

So here's a breadth first search. And you can see, it's sort of parallel pursuing paths. Like it's doing a little down here and a little down there. And it's jumping back and forth between things that have the same depth. I'm going to show it again and I want you to pay attention to the colors, specifically white and black.

You see that it never goes back and updates a point all at once. It does it bit by bit. So you see areas are gradually lightening, rather than lightening all at once. So I show this one more time. This is breadth first search on this maze computed in real time. I can change this maze and it'll come out again.

For the explicit purpose of demonstrating the concept, I actually had to go to the trouble of writing a breadth first search. It's not that hard, at least in Mathematica, and I'm pretty sure it's not hard in any language to write something that works. And we'll talk more about that later. But I know a lot of you have already heard of a lot of concepts in navigation. And you've also got Wikipedia. You could just go online and find, oh, I see. These are the things that are important for that.

So let's move on. Now we have a bug navigation. Bugs apparently find their way round rooms and things by walking until they hit an object, like a wall, and then they just follow along the wall. There's also a well known truth, or truism, or something like that, where if you're in a maze, and you put your hand on one wall, and you follow that wall, then you'll find the end of the maze.

So let's try to watch that at work. Actually, let's make the speed 0.05. So here we've got a bug, and he's walking along. The yellow represents his left hand. No, the yellow is his right hand, and he's following the left wall. So you can see he's going to round to the left. He's really not finding the fastest path from the bottom left to the upper right.

But the point is, he'll get there. So there you go. He found his way there. Let's show that again a bit slower. So the goal here is when you program a bug pathing system, it's that the guy has to remember what direction he has been going. Because if he is just in a position, just any given position-- so I'll pause it at some point, say here. So maybe he's here.

How is he going to know whether he was going down or up? That's the way that he remembers his current state is the direction. Now, that's like a big thing about bug is that it doesn't need any memory at all. Its memory is sort of encoded in its position and its direction. And that's all it needs.

Whereas other forms of pathing, you've got to store these lists, and how long they are, and where they go, and which lists are their parent lists or parent nodes. But bug is very simple. Again, it's not really that fantastic at getting you where you need to go. And you might also ask the question, what do you do when you're not in a maze?

So let's go ahead and answer that question of what you do when you're not in a maze. And we can go over the details of how to implement bug if we have time at the end. But you can look all that stuff up. So just in brief summary, we've looked at depth first search, breadth first search, and bug pathing. I'll talk more about the details of each one, but I like to just sort of go through once, and go through again with more detail.

So let's change it to an alternative maze. This isn't what you're going to have for battle code. That would be terrible. Mazes are not that exciting in terms of video games. Here is an example of the kind of map you'd have for battle code. And what depth first search is trying to do to locate the shortest path.

So first it's going deep. It's going to keep going on its path until it finds the end. Now, there are ways of making depth first search where it won't do what it's doing now, which is kind of silly. What is it doing? Filling space with this line. And all these possible combinations of filling space with this line.

It's just wasting our time here. It wants to go from there to there. What's it doing over here? So that's an example. We're going to make this run as fast as it can run. So let's pause the evaluation and run it again.

As so you can see now it's faster, but it's still trying a whole bunch of combinations that just don't make any sense. Here it goes along the left. Here it goes along this

side. It's going to try verticals. OK, now it's going try all these. OK. Sweep, sweep, sweep, sweep, sweep, sweep.

Now, let's hope, for God's sake, don't try-- oh god, it tried there.

[LAUGHTER]

PROFESSOR: Well, OK. Now that it's tried this area, let's hope it doesn't go keep trying these different things. Like up here, let's hope it doesn't go back there. That'd been such a waste of time. So here it's going-- oh, what is this? It's going--

[LAUGHTER]

PROFESSOR: It's looping around and spinning. And now, it's-- oh, finally. Man. I'm going to fire that guy. Let's see if breadth first search can do any better. Let's keep the delay at 0.1 and see what happens.

Oh, let's see. It's searching the nearby paths rather than going down one that's deep. I see. Doesn't seem to be crossing over old ground too much. So if it's located this point, it's not going to try 10 different ways of getting to this point. One of the features of breadth first search is that if the distance between nodes is uniform, like every square is the same distance from every other square, then as soon as you find the end, you've found the shortest path to get to the end.

So what we're going to see is that it's not going to keep looking after it finds this tile. It's just going to stop. Bing. OK. And it's all done once it looked at that depth. Wow, that was pretty convenient.

Let's see how bug does on the same map. Now, this is just showing the last result. Let's see what result bug gets. It's not going to look like this. So there it is. It moved along and followed this wall. And then it stopped bugging along the wall, and then it followed this wall. And now it's bugging along that way. Are we in a loop? Nope. No, we're not in a loop. It's following along this way. And bing. It got to the result. Pretty exciting.

Let's show that one or two more times to explain what's going on when it's trying to do this. As soon as it hits a wall, bug has to decide if it's going to go left or right. Later on I'm going to talk about why that can end up being a huge problem. You'll see it left the yellow ball behind. I only update the position of the yellow ball when I'm in bug mode.

So the point of bugging in an open terrain is that you have to be able to leave the bugging. You have to be able to stop bugging at one point and start just branching off. Or else, for example, if the goal were somewhere away from a wall, you'd never get to it. Generally, a criteria for leaving the bugging condition is if you are closer to the goal than you were when you started bugging.

I see a question.

AUDIENCE: Why at the very end, and also before it turns on the second wall, does it turn inward?

PROFESSOR: This one here?

AUDIENCE: Yeah.

PROFESSOR: Right now, when the bug is not connected to a wall, I have it going in the direction that is toward the goal. So here, this is like the worst possible map, because just when it wanted to go toward the goal, it hit another wall. So that's why it turned inward. And that's also why it turns inward over here and over there. So that's a very good question.

So let's watch it again. And what I want to point out as well is that the yellow is places it wants to turn. So here it wants to make a right turn. And here it leaves bugging because it was closer to the destination here than it was when it was farther back over here. So once again, it's moving along, and it'll go around and it'll find the answer.

So it's not very good looking, but if your obstacles don't connect with the edge of the wall, it find you sometimes the same result that you would get otherwise. So I'm

going to open up something like Word, and I'm going to make some notes. Because I can type faster than I can write on the board, and that makes a lot of sense.

Let's talk about metrics of success in pathing. There's the time to arrive. You want to have a narrow distribution of times to arrive if you were pathing a group of units. Like if you're trying to get to the other side and you really want them arriving in a conga line, that's potentially going to get them killed one by one.

You sometimes want to explore the map as well. And there are ways of manually working out some of these things. What's the best path to explore the map, and so on. But these are just things to keep in mind.

And let's look at sort of a comparison between bug a and depth first and breadth first search. So the bug system is computationally fast. But at the same time, you've got your own trade off between byte codes and compute time. It uses the unit turn, and you can do that in two ways with battle code.

You can do it with Direction.rotateRight, which will rotate that direction 45 degrees. Or you can do it by using, as we did in the example yesterday, Direction.values, which gets you eight direction values starting with north and going all the way around to northwest. And then you can increment this index.

So if you're starting it direction east, that would be two, because it's north, northeast, east. So that would be two. And then you can just add one to rotate right, add another one to rotate right again, and so on.

It's pretty interesting that bug can work blind in battle code. Usually when you do pathing, you want to do something like senseGameObjects so that you could say, all right, I want to senseGameObjects of robot.class. I want to sense that type of game object. And I want all of them, because I want to make sure I don't run into any of them.

In previous years, you would have also had to do senseTerrainTile. Yeah, it would be something like for a given map location, you could sense-- yeah, it would be like rc for robot controller dot senseTerrainTile. And it would be a given map location

and a or something. And you'd put it, and you'd say, is this equal to TerrainTile.terrainType.OFF_MAP, or it would be VOID, or it would be like OPEN, or something like that.

And it was a huge bother. So we got rid of void spaces. So now you're basically trying to say, I need to use the enemy mine locations. And I believe there's a senseEnemyMineLocations or some method like that. But you don't need those. Because I said the bug is blind.

You could just get by with canMove. And that can be pretty nice. If we're not making giant mazes, you can probably just get by using canMove. If you can't move, then rotate in some direction. That may also end up being cheaper computationally than using these methods for the same bugging. I'm saying bugging with canMove or bugging with senseGameObjects. For this kind of stuff, I always find it useful to go into the release where I've got the folder where I've installed battle code, check my method costs.

I say, all right, let's see here, canMove. Here it is. Costs me 25 byte codes. How much does it cost for me to senseRobotInfo? Because I'm going to get robots, I'm going to have to sense their info in order to get their location. That also costs me 25, but that's after the costs associated with sensing all NearbyGameObjects, which costs me 100, plus all that other stuff I've got to do to sort of set up a loop with checking, and savings. . No wait, that's a different thing.

So yeah, we've got a blind-- it's blind. You should check your leave-bugging criterion. Can be something like patience-- just after a certain amount of time, give up. Distance-- are you closer than you were before? Direction-- how many times have you turned left and right? If you keep track of that, then it can say, all right, I've gone around a barrier. I know that cause of the number of times I've turned. So now I can be intelligent about leaving that bugging.

So finally, with mass pathing using bug, you should consider distinguishing between off-map, void, friend, and foe. And here, void is more realistically said to be neutral mines. I've said before but I'll remind you that you don't know where enemy mines

are until you've stepped on them. So that's sort of a difficulty.

Neutral mines are more realistic to path around. And you could consider writing a rush player, which locates the optimal way to get from one base to another. Because you know the bases. You know where they are. You know where the neutral mines are to begin with. So if you have some of your robots, for the headquarters for example, compute the optimal path between your main base and the enemy base, you could rush there more quickly than you would otherwise.

It's not really going to work with enemy mines though, because as I said, you don't see them. So you might want to distinguish between these different types of obstacle. Because if a foe is in the way, do you want to path around him, or do you want to just stop and start shooting? I mean, of course, in this battle code, you don't have to explicitly shoot. And you move and you shoot simultaneously.

It's worth mentioning here that you shoot after you move. That's something that isn't entirely obvious. You might say, OK, it doesn't matter whether you shoot before or after. That's just a little detail. But it ends up being super important, especially what we're going to see later.

What we'll see is that, let's say one robot of yours and one robot of the enemy's are nearby. If the enemy moves toward you, then it will immediately do damage. Let's say that you subsequently move away from the enemy because you want to, say, rejoin your group. Well, you will move away and then not do damage. So the two of your robots will move along in some direction, and you'll take damage every time, and he won't take any damage at any time. It'll be no good.

Another thing is that if you have neutral mines to path around, well, those aren't going anywhere so you better path around them. But if you have friends to path around, maybe you should just wait for them to get out of the way. Because presumably, they are also pathing somewhere, and so there's probably just a traffic jam. And does it make more sense for you to go running off in another direction, or to just have some patience and wait there for the friend to move?

At the same time, the thing that I described the other day surely applies here, where you have a wall of enemy units here. And you've got me. And you've got a bunch of allies here. And I want to be here at my ice cream. But I can't get there because allies are in the way. And I'm waiting for them to move, and they're waiting for me to move, together with all these other guys that are all boxed up behind this chunk of mines. So yeah, you could realize you'd get in some trouble. This beautiful diagram, by the way. I save so much chalk that way. I tell you.

Let's see. So now we talked a little bit about mass pathing. I'd like to mention one or two other aspects. You could leave room for a bend. And one way you could do that is you could say, all right, I realize there's a corner here. And the first guy that goes around that way will mark this as the waypoint. Then other guys can go straight to the waypoint. Or if they're even smarter, they'll try to go around the waypoint.

So for example, this guy could say, he could mark all of these positions as undesirable. And they could radiate a force outward so that every other robot is being repulsed by these x's. And at that point, you might start to have the situation which would be emergent behavior, where your robots are starting to intelligently sort of have multiple lanes around the obstacle, rather than having just one lane and getting backed up behind one another.

So here they're getting backed up. And here they would be sort of fanning out in a sort of pretty way. Yeah. And you're going to see later the effect of fanning out. I mean, of course you've seen it, and you can imagine how strong it would be on the outcome of a battle. But that's going to be really important when we talk about the emergent behavior of your army when you take into account local information about where people are standing.

I hope you guys like the analogy that I sent in my email. I thought it was a little bit contrived, but not too terrible. Not too-- OK, it was pretty terrible. OK. So in mass pathing, also consider your execute order. As we've mentioned before, if you have guys in a line, and their robot IDs are 1 and then 2 and then 3 and then 4-- yeah, this is real pretty-- then they can move left because they'll execute in this order. Oh,

man. It's like it's animated. Ah, jeez.

But they can't move to the right, because this guy's in the way. So number 1 will be like, I don't know where to go. I'm going to go there. Number 2 is like, uh, I can't go here or here. Maybe I'll go up there. And he gets lost. His head's cut off.

AUDIENCE: Can 2 move diagonally?

PROFESSOR: Yes, 2 can move down here. There is diagonal directional movement. Yeah. And 3 could move down there. And then I guess 4 could move here, because he doesn't know that the others-- it'd be kind of cool if they did that. And then they would just like shimmy along. It'd be like an inch worm. That'd be pretty cool.

Oh, gosh. Paint, don't do this to me. There, that's much better. So they could do this, but 4 would have to be pretty darn smart. It's more likely that by now these guys are inch-worming along, and 4 is over there. And eventually, 3 gets sort of spread out. And I don't know, 1 just goes and he gets tired of the whole deal because he's been looking at 2's behind the whole game.

So yeah, you definitely want to keep in mind the execution order. And when I say keep in mind, I mean you could actually do sort of like a bubble sort, where if your ID is different from the guy who's behind you, you could switch places with him and try to get behind him. Because you can use the function my robot controller dot getID, and you'll have your ID.

Remember when we did robot, and we said, all allied robots, or something, equals rc.senseNearbyGameObjects, the most helpful function name in the world, and we said robot.class? You could say, all right, all I'm interested in is the guy behind me. Please give me the ally guy behind me. Well, you don't need to get all the nearby game objects. You could say, all right, I want the one who is adjacent to me. So he's within a distance of one, or say two if you want to include diagonals. And then I also want to make sure that he's on my team. So I do rc.getTeam.

So now immediately, I've got a nice list of allied robots that are nearby. If I want to get their ID numbers, it's as simple as rc.senseRobotInfo. And I sense one of their

info's. This is aRobot, which I can pull out of the list by doing something simple like I'll locate the closest one in the ordinary way that you would expect.

So now I've got the robot info, and robot info includes-- Actually, I think you might even be able to get ID from robot. Where would I go to check? I would just simply go to the documentation in the release, all classes-frame. I would see robot info, just try to see. Or maybe let's see if the robot has-- oh, robot has getID. Look at that.

So I don't even have to senseRobotInfo. I can just go straight. So here I can just say, ah, I'm just going to aRobot.get-- what was it? Get ID? I think it was either get ID, or ID. And now I'll have that ID, and I can compare it to my own, and so on, as I said.

So this is all about bugging and mass pathing in a simple way. If you want to do it smarter, you can use a*, which is a breadth first search. You can look up all Wikipedia. And it's making a list of where you can go. That's as simple as it is. You're making lists, and you're keeping that list pruned down so that it hasn't got all these extra places for you to look at. So it's shorter to get to your goal.

And what it can also do is incorporate heuristics. We'll be talking a little bit about heuristics today, where we're saying it's a fancy word that, to the extent I understand it, it just means that the distance between one square and another is not necessarily one. Or the goodness of being in one square or another is not necessarily the same.

So maybe you want a path from point A to point B, but you don't want to cross a particular obstacle. I saw once that somebody did a giant paper. I think they did a Ph.D. Thesis on the traveling salesman problem. And they had solved the traveling salesman problem under the condition that they were traveling in the United States, and they did not want to cross the Mississippi River.

That's so beyond-- why would you-- I love crossing the Mississippi River. But anyway, he didn't want to, so he did that. And I think he went ahead and got-- now

that I remember it, it was actually a high school science fair. Ph.D., science fair, same thing.

So you can incorporate these heuristics, and it's just a way of saying, here I've got this list of locations that I'm going to. And I'm just going to associate goodness or badness with them. And at the end, instead of saying distance, I'll just say goodness or badness. It's as simple as that. And that's how those things end up getting done, and so on.

There's also d* and a whole plethora of other things. This is like start from the destination and go to the start. There's various reasons to do this. There's tangent bug, where you're saying, all right, I recognize a certain pattern to a* and d*. If I start at this location and I end here at e, and the obstacles sort of look like this, well, the only reasonable places for me to go in this open continuous terrain is tangents between the two things. So I can draw a tangent there. I can draw a tangent there. From my starting position, I draw a tangent to there. You see these tangents? They're right angles. It's beautiful.

There's one here. OK, that's a right angle. This guy's got a tangent here, and he's got another one there. There's some arbitrary dumb ones that never apply, such as this one. Or like this one. Well, in any case, there are ways of pruning it so that you've got just the right tangents. And then you've really only got like six paths to look.

You can go here, and then there, and then there. Or you can go here, and then you can go from-- ah, yeah. There's one. You can go here from there to there. And then that's like a really dumb way to go there. So it's really easy once you have tangent bug.

And there's a way to generalize that so that it's not just vector geometry in open space, but is down to the level where you're like, OK, these are individuals squares. I zoomed in to indicate-- that was really smart. It worked really well. I was happy with it.

So there's tangent bug, which you can use there. And you could consider having-- in tangent bug especially, you could imagine that a headquarters could help out. Where you have these things that you need to get around. Oh my goodness. Anyway. You want to get around them so that you could have the headquarters indicate what is the tangent spot. And he doesn't have to tell you all the places. They're just like waypoints to go to on your way to the destination. And it could work pretty well. It could. So keep in mind.

There are some undesirable behaviors to watch out for when you do something like this. Sometimes allied units that bug in the opposite direction, that makes a lot of sense if you have a little obstacle to get around. But if the obstacle is big, sometimes your robots will come at one another from opposite directions when pathing.

So you'll have-- here is the obstacle, and Robot a hits the obstacle and starts going this way. Robot b hits the obstacle and starts going this way. Robot b is here and a tries to go around him. b keeps going this way, and a now is like, I don't know where I am. I'm just going to spin around in midair. Because I thought I was bugging, but then the wall changed.

I know that there's like a million movies about how people are in mazes, and then like the big kicker is supposed to be that the maze is shifting over time. Well, by now we're used to it, and it's boring in a movie. But when it happens in battle code, it hits close to home.

Another undesirable behavior could be that you bug around the whole map. And one thing you can try to do sense if the terrain is off map. Again, that would be TerrainTile.TerrainType=TerrainType.offMap. And you could check that, but sometimes there'll be like, maybe we'll put neutral mine's all the way around the edge of the map. And then you won't really know if you're bugging around the edge of the map because it'll just look like neutral mines. So there is another thing to worry about it.

You could turn around a transient obstacle, like an ally, as we mentioned before.

There's insufficient information for a lot of things. Like maybe you've got the perfect a* algorithm that's going to find everything. That's the best technical term for it. And you're going to find everything, but you don't have that much information. Like you don't know where the enemy mines are. You don't know where the enemy robots are until you get close to them.

And so that I think that's fairly well explained. We've got everything laid out on the table here and there. I would mention that here's another use for radio, is not every robot can tell that another robot is stuck. So you might consider radioing information about robots that are stuck so that the other robots, allied of course, behind them can get out of the way. And you can make this playful in your own way by coming up with your own comments on what-- anyway, that made a lot of sense. So they can get out of the way.

So let's consider the following case. You have this radio system which consists of these integers. And you need to be able to post map locations to integers. How are you going to do that? Well, I've got a good idea for you if you haven't thought of it already.

An integer in Java can go from something like minus 2 billion to 2 billion. So here we're radioing integers. And it has to be an integer. Well, I know a certain thing about 2 billion. 2 billion has this many numbers in it-- 1, 2, 3, 1, 2, 3, 1, 2, 3-- that's a billion. Yeah, I got it. I think that's 2 billion. I think it has 2 billion because it's 2 the 32. Yeah, something like that.

Well, I know what this can contain. All I have to do, since a map location starts-- it has x and y values starting at 0, 0 and going to, at most, 100, 100. And something tells me that the final maps we give out won't be that big. But maybe they will. I don't know. They might be.

Well, look at that. 0 and 100 only take up three of these positions in the number. I mean, you could do it in binary and encode the information even more efficiently. But this will do just fine. So here's a thought. Put the round number in the first three, put to the x-coordinate in the second three, and put the y-coordinate in the third

three. And now you can tell when you look at a single integer if it is up to date. So here, we'll have 2, roundNum, xval, yval. And all you would do for that is you would say, my integer equals round-- it would be-- let's be explicit. Clock.getRoundNum()*1000000+xval-- it'll be my map location, which I'll call m.x*1000+m.y*1. There you go. And that would encode everything into my integer.

And then I could make another function that sort of unpacks things into a map location. Oh my goodness. What would that look like? Well, it would be something like public static, and it returns a map location, because that would be really useful. Public static MapLocation, we'll say intToMapLocation. Yes. That will be really useful. And it takes an integer, myInteger. Yes.

I'm coding in Microsoft Word. I wonder if everybody has done that. And if you're going to tell me it's Open Office, that's true. That's true. I acknowledge that. Let's see here. So let's get the other values on out. Let's say that our x val, which is going to be an integer, equals myInteger at-- ooh, what am I going to do here? I'm going to say, I'm going to take modulo 1 billion. Is that going to do it? I'm going to put spaces in so that I can see it. But I think if you actually put spaces in, it isn't going to work.

Yes. So is that my x val? No, that's my y val. So I am going to do this in this order because it'll make it easy. int xval=myInteger%1-- you guys got to pay attention. This is the 1 billion, isn't it? Yeah, I've got to make it this way. Actually, I think they might be identical.

Anyway, if I do it-- yeah. Uh, no. This has to be billion. Sorry, that has to be a billion. Because if I do modulo a billion, then any numbers that are over here sort of roll off from the 2 billion. Yeah, that'll be fine. And I'll just subtract from the millions category-- would that be 1,000? It'll be a million. And I'll subtract off the y value. And now I have an x val.

And finally-- I mean, maybe the first thing I should do is I should check to see if my very first number, like if these very first three numbers, I should check to see if those are decent. I here some whispers, but I'm sure that they're talking about how

amazing this is going to end up being.

We'll make a new map location by saying, the output map location-- well, I'll just call it outputLoc-- MapLocation outputLoc equals-- now this is a constructor-- new MapLocation-- this is the same thing we did yesterday-- (xval,yval). And I could have a check in here that verifies that it was set at the previous roundNum, so that it's not out of date. That'd be really useful.

So that's a pretty useful thing to have right there. That's just one of many byte code tricks. So this is going to be packing a map location into an integer. What's another one?

Let's consider that you have a whole stockpile of energy that your robots are using. But early on in the game, you're wasting a lot of that energy because it's just decaying in your inbox. It's just sort of floating around, and then it's going down. So why don't you consider spend byte codes when the number of allied bots is low? Or equivalently, when there's a lot of rc.getTeamPower. Question.

AUDIENCE: For the 20% loss every round, is that your total stockpile of power, or is it the total increase?

PROFESSOR: It will be the total stockpile. That's right.

AUDIENCE: So [INAUDIBLE]?

PROFESSOR: That's right. You'll lose it. So that's-- what's the common saying? It's use it or lose it. Yeah. So if you're using it, you don't lose it. I guess that was not necessary to explain.

I made a program here that may be useful on this effect. I call it stockpiling. And I made it in Mathematica just because it's nice and visual. So here's an example of stockpiling resources. Let's go ahead and we'll simulate 400 rounds. We'll simulate the case where the decay multiplier is 0.8. And we'll start with zero bots, using zero byte codes.

So you can see here, this is what you should expect your number of bots to be.

You're building them constantly, and you finish building them around turn 300. And you hit a maximum of 40 bots. If you're using more like-- let's do a little bit more realistic-- you're using 2,000 byte codes, well now, you're maxing out at 34 bots.

And you can see here that you quickly decay upward to your maximum near 200, and then you only go down bit by bit as you build more robots. You may, for example, have-- I mean, we can look at more examples here. And I'll probably talk about this more on a strategy lecture, but let's say that you got the fusion upgrade, so that the decay multiplier went from 0.8 to 0.99. So now we've got 0.99 there.

Well, now you can see that you got up to 44 robots because the decay was much slower. So you got up to a really big number as you were massing up these robots. And so for about 70 rounds, you were able to maintain an increase of four robots over the enemy. Not a huge benefit, if you ask me.

Now, if you stockpile for longer, let's see what effect that would have. I guess this is a bit of a tangent, but we just did tangent bug, so it's relevant. Ba-dum! Let's see here. So where's the-- oh, yeah. Number of rounds. Round is greater than. So we're only going to start building robots at round, say 200.

Now we can't really see what's going on. Let's increase the number of rounds in the simulation to 1,000. So now we can see here that we are maintaining-- we're having five extra robots for slightly longer. It's not that great. It's just not that great. Let's wait a little bit longer before we start building so we can try to stockpile.

OK. We've stockpiled as much as possible. We've still only got five extra robots. Oh, actually it's more than five extra robots. Let's go down to zero so we can make sure. Yeah, it's 12 extra robots on top of 40, so it's 12 over 40 times 100 is 30% more. 30% bonus, but you did it at the price of stockpiling for 500 turns. And by then, you'll certainly be dead.

That's a quick look at stockpiling. Question.

AUDIENCE: Is power used for anything except for [INAUDIBLE]?

PROFESSOR: Yes, power is also used for capturing. In my handy dandy unit spec sheet-- gosh, I did use that term. Handy dandy. In the spec sheet, which I put over here on this place, which you can get it on the website. I could be going to the website. I'm sure all of you have been there by now. And if you haven't, then you're missing out.

This useful battle code data sheet, which updates from time to time, indicates what takes power. So if I just do Control-F and the word power, we can see here, the headquarters generates 40 power. OK, we know that. Soldiers upkeep to one to two power per turn. OK. Attack power, that's irrelevant. The generator generates 10 power. OK, I see that.

And then it should also say somewhere that you capture encampments at a cost of 20, plus this number, of units of power. So if you own one encampment and you are capturing one encampment, then the next encampment you go to capture will cost three times the ordinary encampment cost, which is right now 20. So there you have it. That's another use for your power.

And it's certainly interesting to consider that you might need to call suicide on some of your robots so that you'll get enough money to buy some of those encampments. . Because that's a one time cost. So you might suicide some robots, build an encampment, and then build those robots back again. Because it is not a continuing upkeep. Although I do believe that encampments count as robots to the extent that they do require upkeep themselves.

Let's go back and see how much we have time for. I'm sure everybody's mouth is already watering as much as mine is about the chicken curry, among other things. I was over here putting down byte code tricks, and I got sort of sidetracked. Let's talk about more byte code tricks.

So let's think about how it might be cheaper-- oh, my goodness-- to broadcast an enemy's location than to sense it. Let's answer that question. The method cost for read broadcast is one. And for making a broadcast is also one. Now, that's pretty small compared to sensing an enemy location, where, let's see, sensing an enemy location. Sensing an object is 25. Sensing a bunch of game objects is 100.

I mean, 1 is less than 25 oftentimes. So let's check if there any other costs associated with it by going into the specs here under constant values. So here in the Java Docs, we can see that there is also a broadcast read cost. So that also answers this gentleman's question that there's also another cost in power.

This read cost is a power cost for reading and broadcasting. So this is going to use 0.05 power. It's relatively small compared to your one upkeep [INAUDIBLE] positions. Well, then all of a sudden-- or read 20 positions would be 0.2, and that would be equivalent to using 2,000 byte code. So that starts to say that that's a pretty high cost for reading in terms of byte code. Which suggests that if you want to send things on messages and be byte code efficient, that it makes the most sense to send the results of a large computation rather than the base data on which that computation is performed.

Let's see here. It's already 5:49, and I haven't showed you as many awesome matches as I was expecting to. I'll blast through a few more byte code tricks and then we'll get right there. Make sure to check the byte code cost of your functions. I find it very helpful to put in my code a little block comment that says what the byte code cost is. And I'll put it right next to the function, so I can say, all right, that's the price of that function. And how many times can I use it? Do I want to use it? Is it worthwhile? Am I improving it?

One way to do it is to run the match with and without that function installed on your robot, and just do the math yourself. But another way is very simple. You can just use Clock.getBytecodes, or something like that. There's a function there. And you can just do it before and after a certain operation and then check it out.

You should also try computing only relevant directions. Later on I'm going to show you a little example of how you can write code where a robot is doing something in two ways, and it's trying to find out who's adjacent. So it's like, all right, I can move forward and that would put me next to this many enemies and this many allies. I could move to the left and it would be this many enemies and this many allies.

You kind of want to be next to a lot of allies and just one enemy so you can gang up on him. That was the whole idea of being intelligent about your stance and where you are. Because you want gang up on the enemy. And that's a very local decision when it comes right down to it sometimes.

But if you can't move in a certain direction, don't compute anything about that direction because you're not going there anyway. That's a thought. I mean, of course, you may compute it anyway, and then try to move in that direction, and move in the next closest. But you could consider that. And I'll show a little bit.

So you could consider spending things only when it's low. Cheaper to broadcast the enemy using that, and so on. I have another useful thing when we're looking at pathing. And I'll show it by running a match. I'll run the match of Cold War, which is my player, versus Hard Bot.

I have the Cold War player, does something that's similar to the realistic Cold War. But I'll do it on slightly smaller maps so you can see what's going on. Let's see here. Let's go on tiny. So I made these maps that are free of mines just so I don't have to worry about it.

So here we've got Cold War in blue. And his goal is to nuke the enemy. And what he does is he paths around over here around the med bay to defend against the enemy. And the enemy will be just building these shields and then attacking. And they attack my med bays, but they get nowhere. I don't think I eve lose a single guy.

So the Cold War is going to finish massing up a bunch of troops. And when he's done, he's going to build a nuclear missile. There you can see the projects going on. Hello, and thank you. Yeah. And so he's building along the nuclear missile. And when he's done, it will be curtains for the other guy. Curtains, as it were. We will bury them.

But you might ask, I can't really tell if these guys can see those guys or what. You can push the letter A, which gives you site range and attack range. So that's a pretty useful thing to have there. So that's A in the client viewer.

There's also another list of commands, like B will turn off the broadcast range. Let's look at some more options. I would have liked to go into a way to store-- here's another byte code trick. Store the locations of other units in an integer array, so that instead of having to loop through all enemy bots and check if they're adjacent, for example, you can just store their locations into an array so that you know where they are. Then you can just index into that array. And you say, all right, this is where they are.

So you're indexing into a two dimensional array. And I can talk to you about how to set those up at another time, but the result is that you can perform significantly better than you otherwise would. So here, Player 3 kind of works. I'm going to show it where doing this calculation, where you find the number of adjacent units can be pretty helpful.

So here's what I'm showing. On the upper left and the bottom right are the same robot. But what I've done is I've done, if your team is A, then run the code this way. If your team is B, then run the code the other way. And this is demonstration. So let's back up just a bit. I'll zoom in as much as possible, but that's as good as it's going to get.

So let's go until they're touching. So this guy is computing adjacent robots, and he's printing out the number of adjacent robots that are allied. In fact, I think this might be-- yeah, this is the number of allies. Starting with north and going around . And this guy is printing out different numbers that are heuristics.

So for each of the positions around him, he is computing how many allies there are and how many enemies there are. And he's going to try to move in the direction that has the most enemies. So he'll either stay put if his current location has the best heuristic value, or he'll go toward the best position. Like if he's going to try to flank the enemy, or get to a position where he'll gang up on the enemy.

And you can see here that the red player is able to make significant advances against what was previously Awesome Bot. So you can see, I'll show again, it's very hard to tell. Because it's a subtle thing that takes place where it's trying to do an

optimization at a very local scale. So you're taking good decisions on a robot by robot basis and having them translate into good decisions on a global basis for the whole team.

It was a bit of a close race, though. So let's consider making global decisions. Here's another pathing idea where you would be using the local data from each robot to decide that you might want to form a concave around the enemy. So let's do that example. And then I'll end with one final example.

So this one is a concave. I have them just build up robots until round 200, so I'll skip until there. And you'll see that the blue team is going to sort of back away. Here I'm going forward. He's backing away from the enemy. And then when the moment is right, he comes in. Boom.

He comes in and surrounds the enemy. And they started with the same number of units, and look at how many he has left. Look at the difference that that makes. So you should really consider having your robots think about what's close to them, nearby, and path with respect to that. I know it's not exactly a* or d* or whatever, but it's something that's worth doing. At this point, the other guy really can't return from that problem.

I'm just going to show you one more example of how things can get interesting. I've changed the constant values all around a little bit, just to show how things would look if both players were the player that tries to avoid the other player. Now of course, when you write your own, you write it in your own way and it will behave in that way.

So it won't necessarily look like this. And there are things about this that are just downright stupid. But you're going to see it happen, and I believe it will be as fun for you is it was for me. So here we're on round 200. I wish I could slow this down, so I'll just click forward. You see they're both avoiding one another. And just there, the whole enemy team was backing away from one guy.

[LAUGHTER]

PROFESSOR: Why? That guy was moving toward the enemy I think because he doesn't see them. They move first and then he moved. So he can't even see them. He doesn't think he's that close, but they think they're close. Because this guy is Robot 138, and this guy is Robot 139. Can you believe it? They're that close. Oh, my goodness.

So they're going to run away from one guy. Meanwhile, these guys are running away, but they're also sort of edging that way. Because they figure that's where the enemies are because it's the enemy headquarters. These guys are running away. And there's sort of self-segregating into these separate packs. I don't think a single bloodshed has been spawned. That was the right way of putting it.

And they're like wrapping around one another. They're going to trade sides of the map. This is absolutely bonkers. So you can see here that the red is pretty much going to abandon his own headquarters. Blue is going to abandon his. Because they just-- it got smelly, tiresome. They want something new in their lives.

And when we continue on to the end of the match, it's just a question of which one is going to be able to deal the damage sooner. Like this guy is saving himself on the basis of producing units bit by bit. He's like, oh, if I stop building units for one second, I am dead. I am totally gone.

[LAUGHTER]

PROFESSOR: Keep building units. Wait, no. These guys are eating up all the resources, and these are dying because of the end of the round. That is, they don't have enough power. No, please don't eat all my units. Please don't eat my resources. You guys should suicide. Protect the headquarters. You're doing it wrong. I swear, you're doing it wrong. No, no, no. You've got it all, but now we're killing his. And we're doing a lot, but oh, we've taken a lot of damage. Ahh, and then it's over.

And so there you have it. A little bit of exciting stuff happening with direction. Thank you for coming. But let me say that one of these dishes, I believe it's the redder looking one, yeah, the one in the middle, chicken vindaloo, this time is spicy. All right? So beware that one. Don't take too much of it, because these guys know what

they mean when they say spicy. Thank you very much.