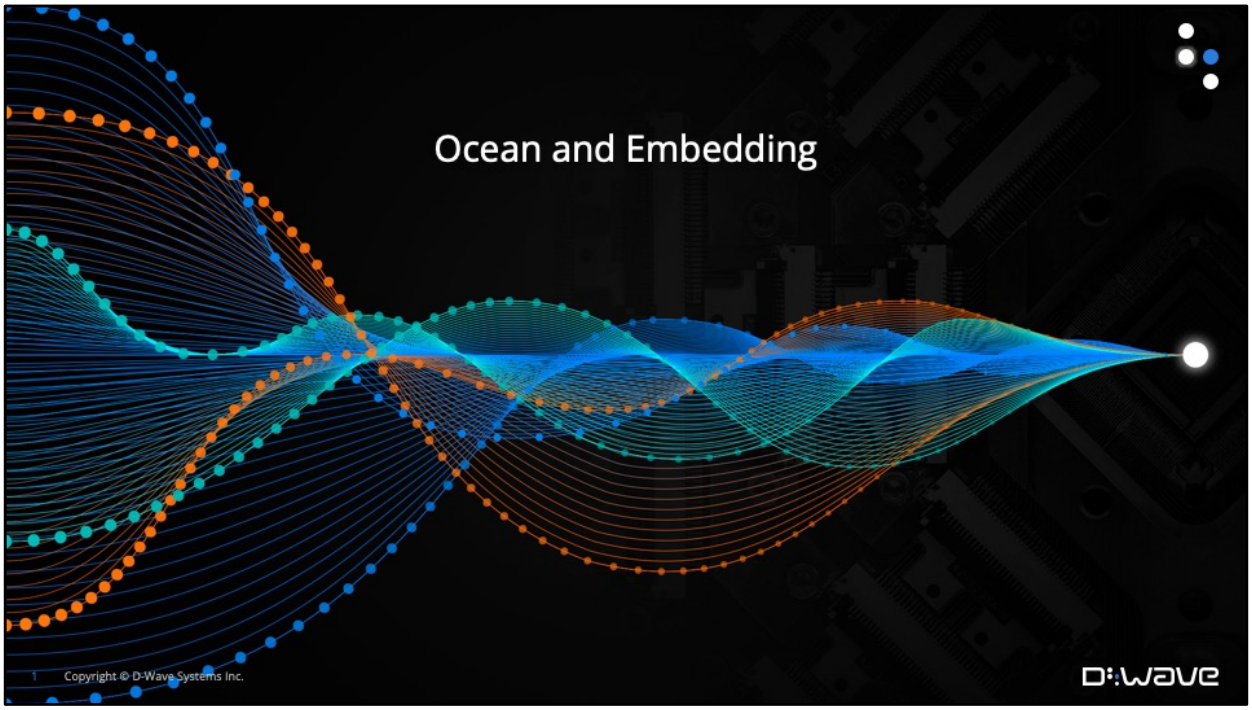
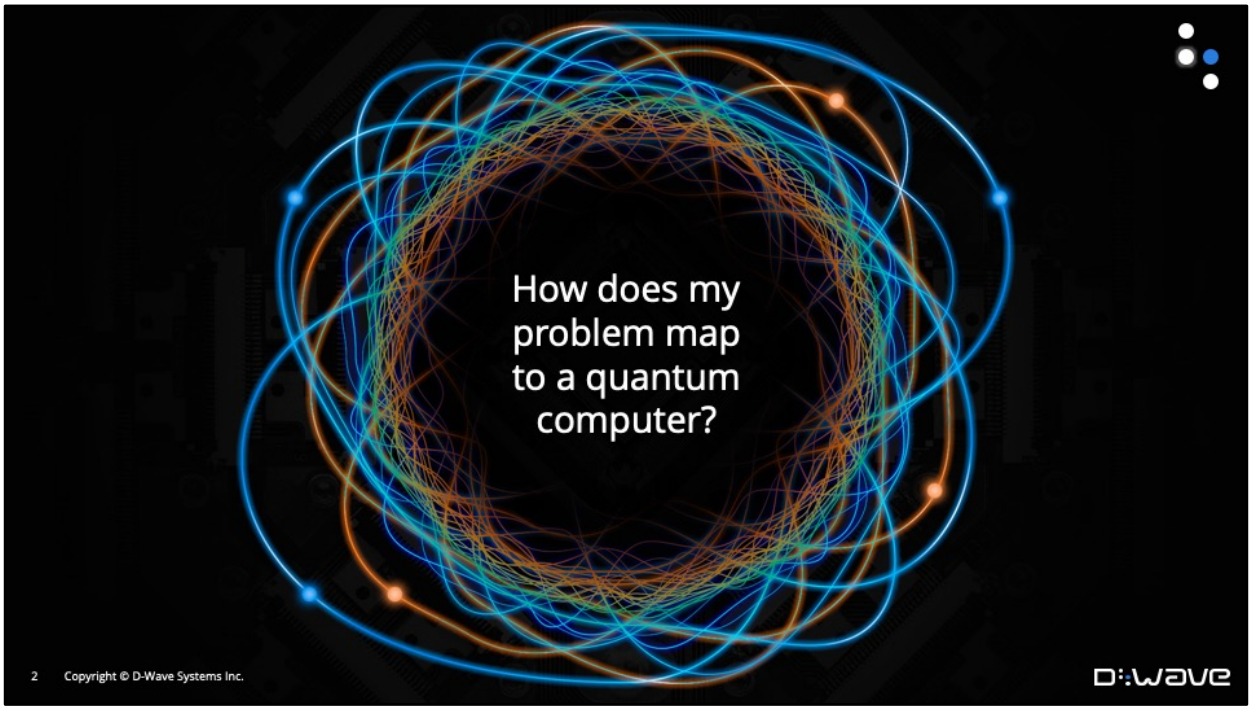
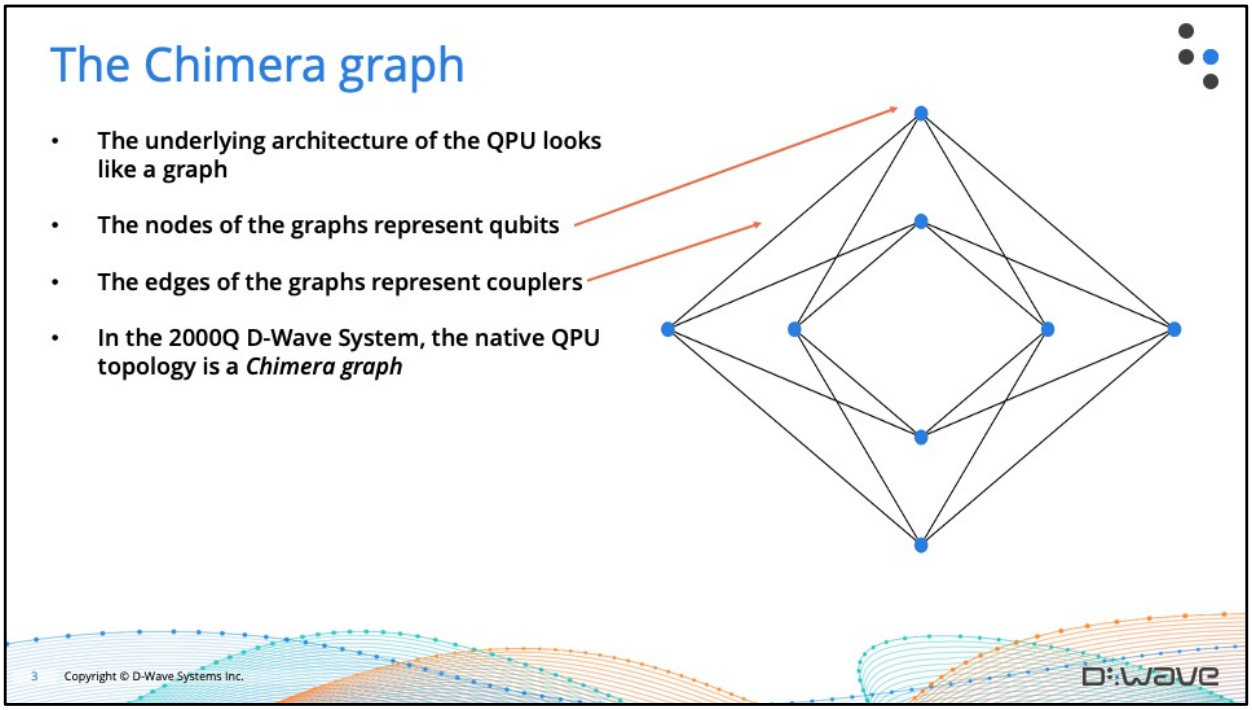


In this module we'll explore how the logical problems we have explored so far map onto the physical quantum computer chip's qubits and couplers. This process is called embedding.



We've looked at how to formulate problems as QUBOs. Once we've come up with a QUBO how does this map to the QPU? How do we map linear and quadratic terms in a math equation to qubits and couplers on a quantum computer?

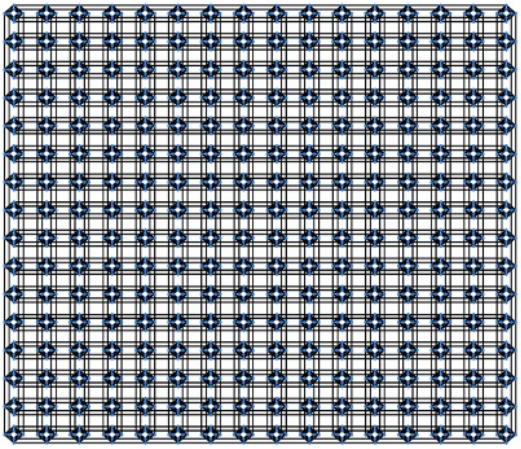


What does a quantum processing unit, or QPU, look like? Under the hood it looks like a graph. This network built up of qubits, or the dots in the image, and couplers, the lines in the image, is what we call the QPU topology. Different quantum computers have different topologies. The one that we see here is for the 2000Q system, which we call the Chimera topology.

We're looking at one small section of the Chimera topology right now.

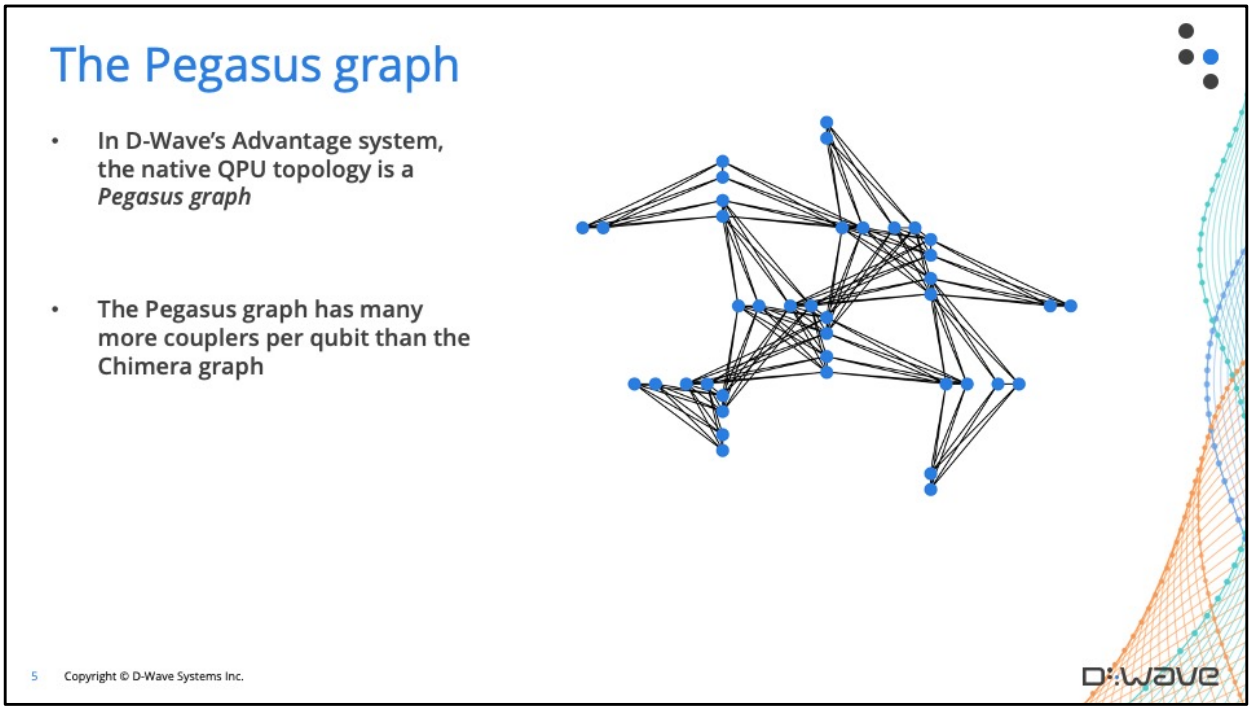
The Chimera graph

- The underlying architecture of the QPU looks like a graph
- The nodes of the graphs represent qubits
- The edges of the graphs represent couplers
- In the 2000Q D-Wave System, the native QPU topology is a *Chimera graph*



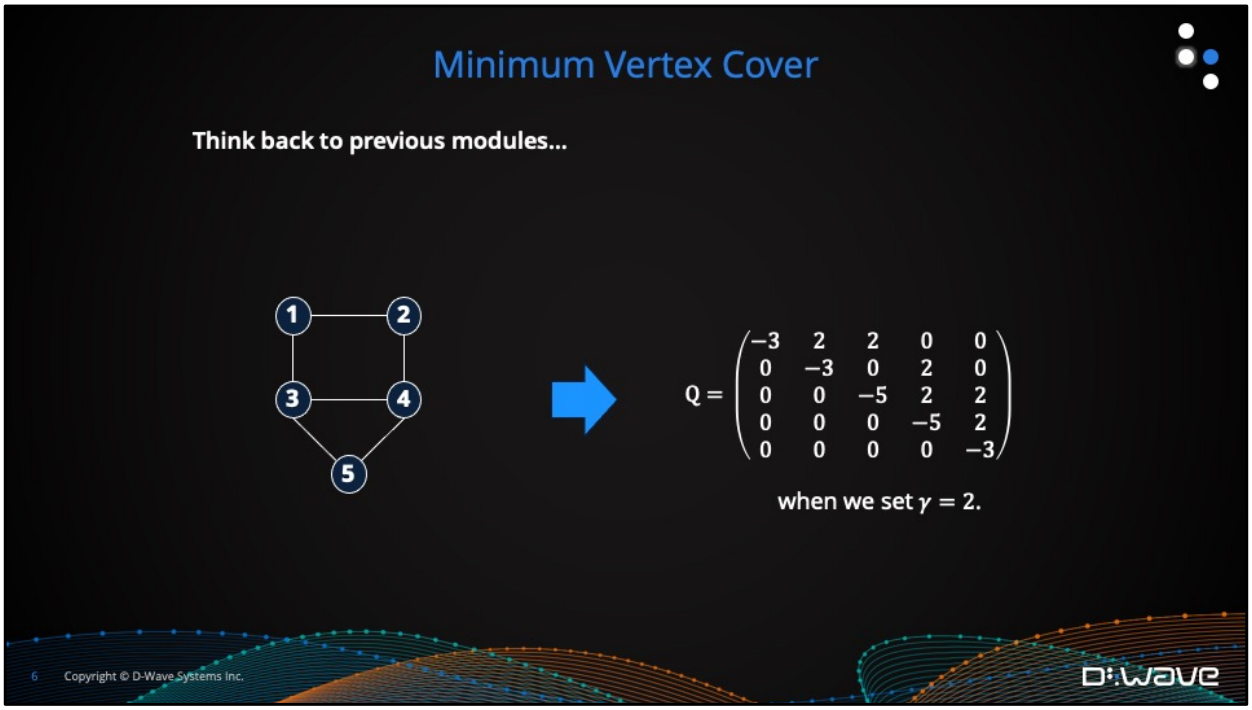
Copyright © D-Wave Systems Inc.

If we zoom out, we can see the full scale of the 2000Q QPU.

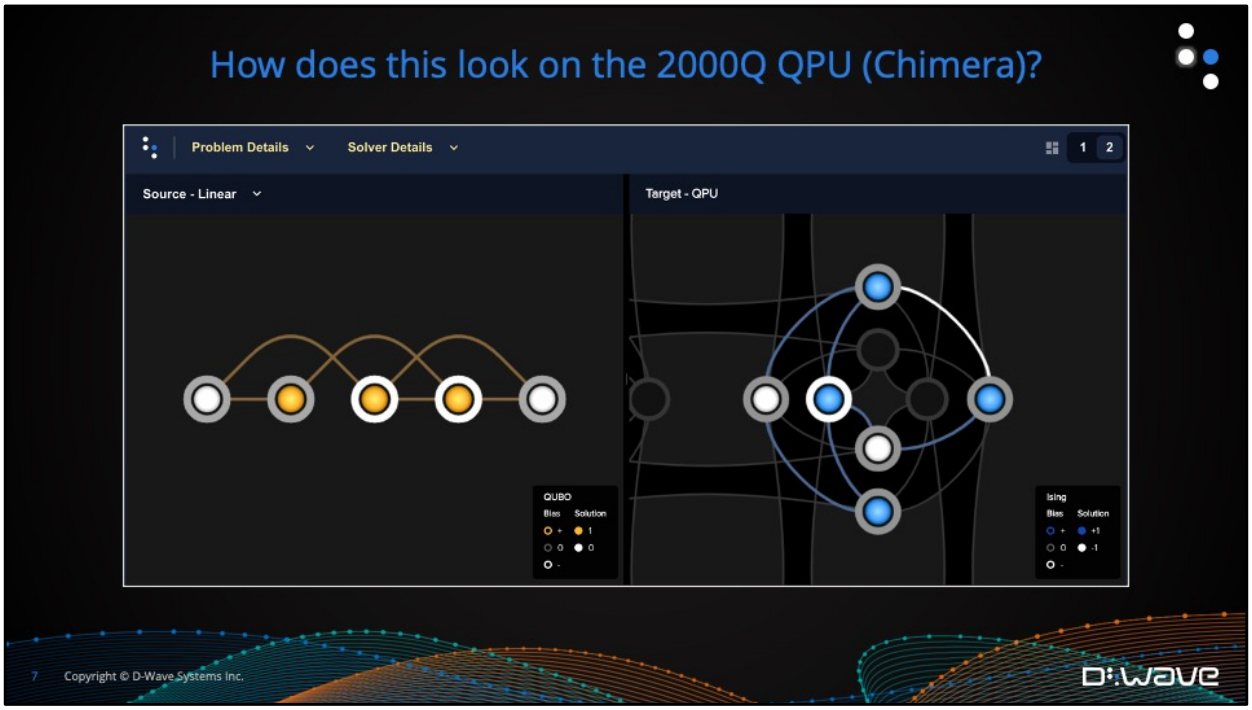


D-Wave's Advantage system has a different topology connecting qubits and couplers. The Advantage system uses the Pegasus topology, which has many more couplers than the Chimera topology. This picture shows a small version of the Pegasus topology, which you can see has many more edges or lines connecting the dots. Again, this is a small version of the Pegasus topology. If we zoom out, we see a much more complex image of this topology.

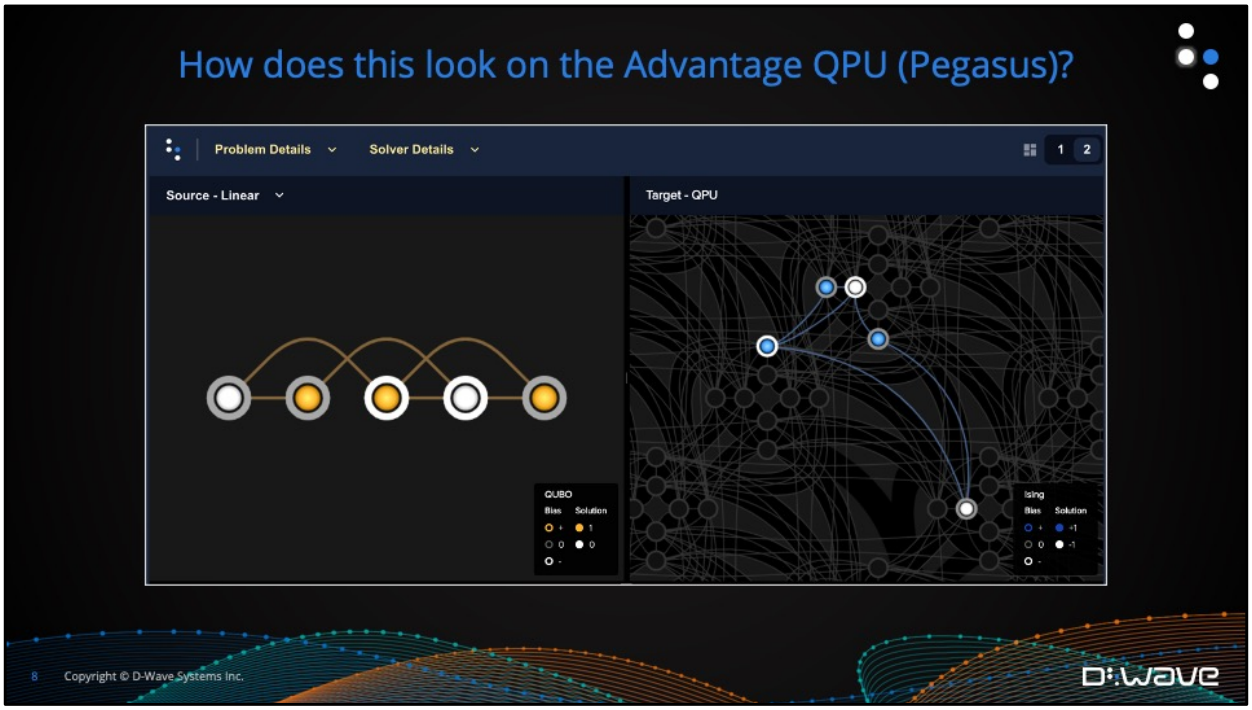
Here's the Pegasus topology with 680 qubits.



Let's start with a problem we looked at in an earlier module: the minimum vertex cover. We started with the graph on the left and formulated our minimum vertex cover problem as a matrix. The matrix we came up with depended on the value for our Lagrange parameter, or gamma. If we plug gamma=2 into our matrix, we get the matrix Q shown here.

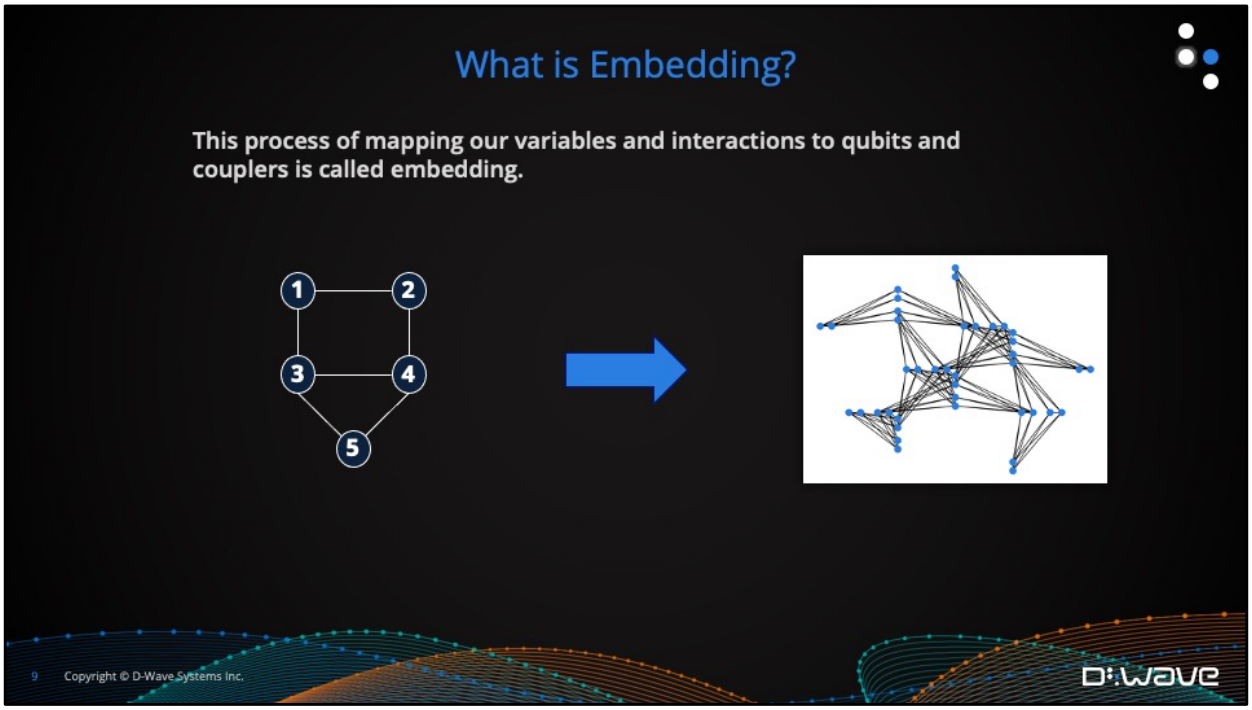


When we run this QUBO on the 2000Q quantum computer what does it look like? On the left is our logical problem – this is the graph representation of our QUBO, just drawn a little bit differently. On the right is how our problem is physically mapped to the 2000Q QPU. Each blue or white circle on the right is a qubit, and a line connecting two qubits is a coupler.

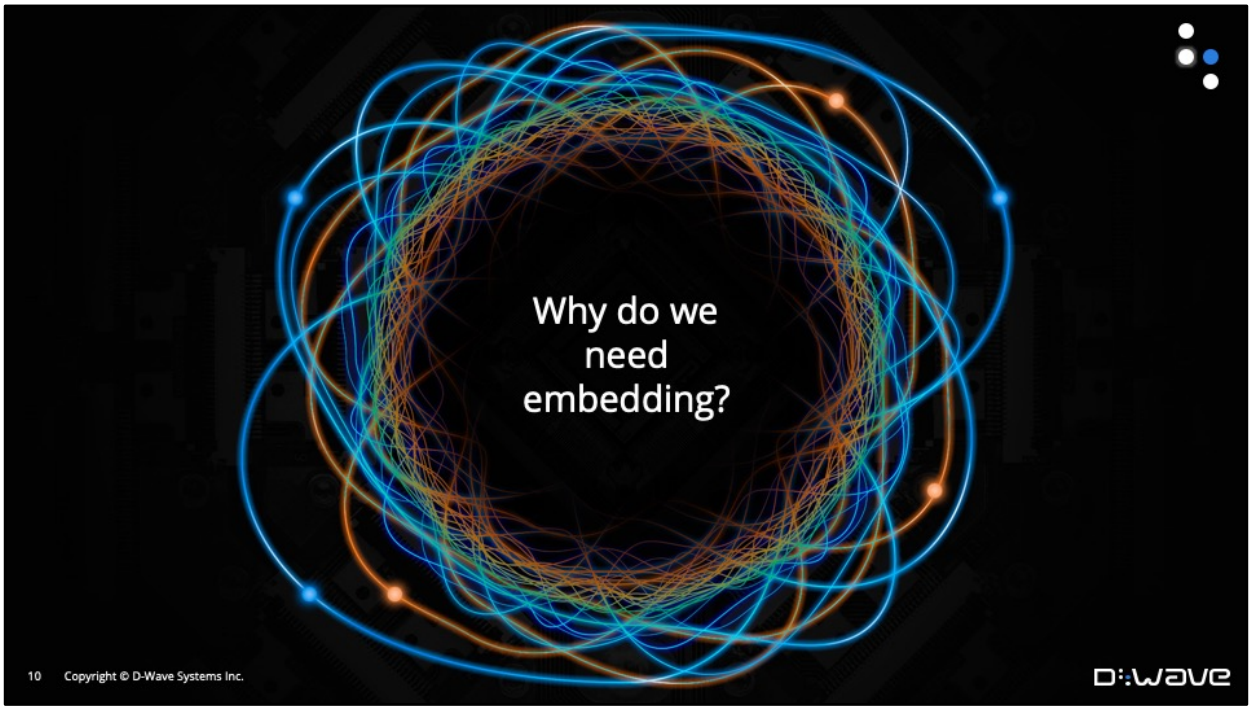


When we run the same QUBO on the Advantage QPU, it looks a little bit different. This is because the Advantage QPU has a different layout of qubits and couplers. The Advantage layout is called the Pegasus topology. You can see in this picture that there are far more couplers present on this chip, allowing us to embed much more complicated graphs more easily.

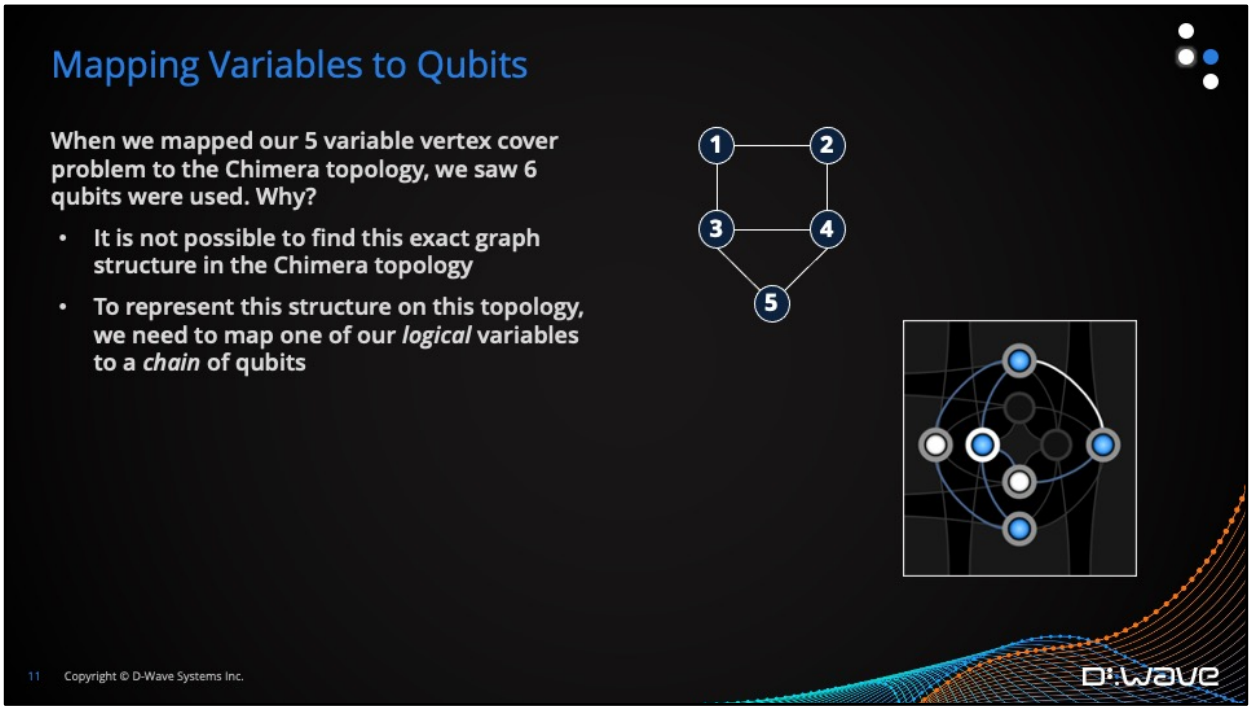
Generally we try to map linear terms (shown as nodes in our logical or source graph) to qubits in on the QPU. Similarly we try to map quadratic terms (shown as edges in our logical graph) to couplers on the QPU. In this embedding shown, each node in our logical graph maps to exactly one qubit on the QPU. The values in our QUBO map to qubit weights and coupler strengths.



This process of mapping our logical problem as a QUBO onto the physical qubits and couplers is called embedding. Every problem that we run on any quantum computer needs to be embedded to match the topology of the QPU.

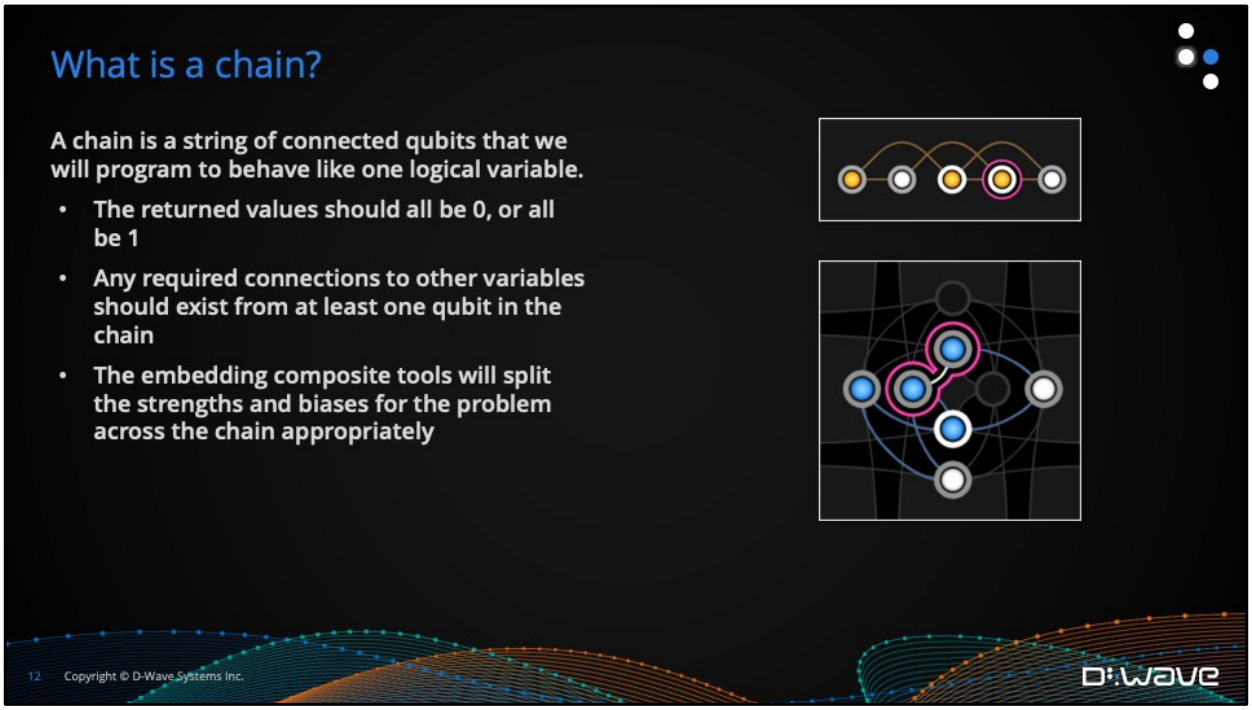


Why is it so difficult to embed problems onto the QPU? What makes this complicated?



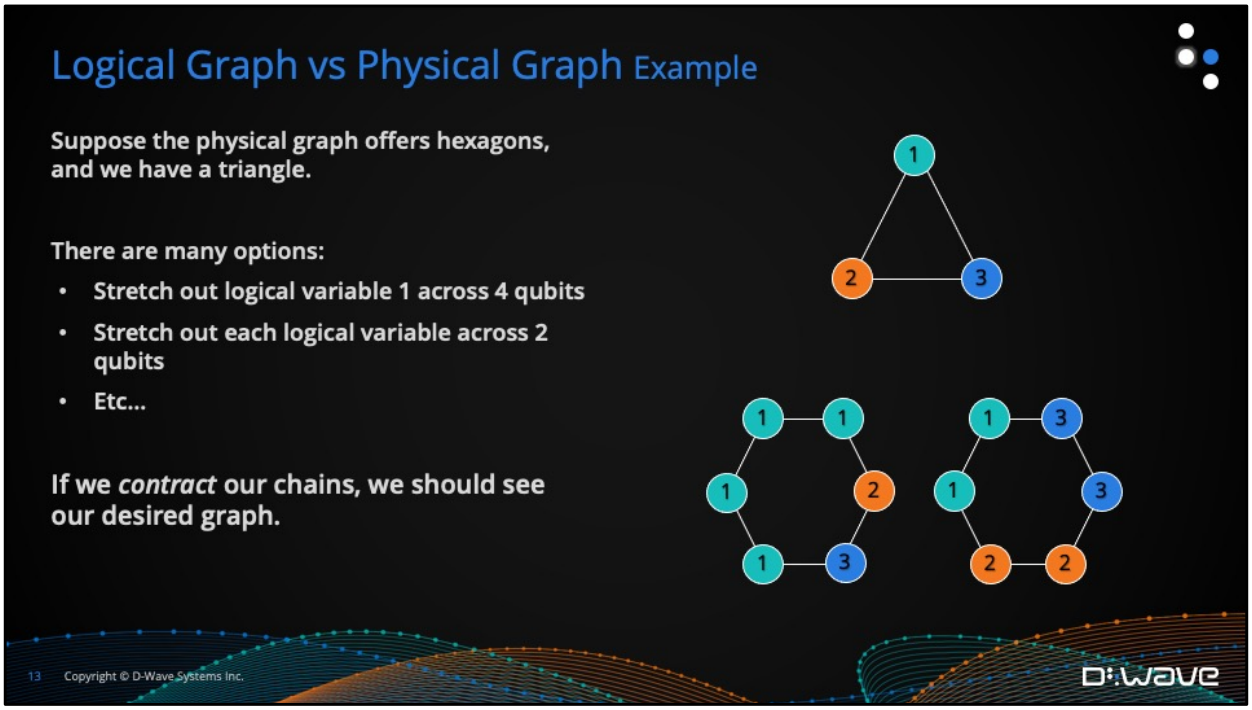
Here's our vertex cover graph again. You may have noticed that when we embedded our 5-variable problem onto the Chimera topology that six qubits were used. Why?

The Chimera topology does not have this exact graph structure. To embed this QUBO into the Chimera topology, we need to use chains. Chains are groups of connected qubits that are mapped to one single logical variable. A chain of qubits should behave like one variable – when our solutions are returned we want them all to have the same value.



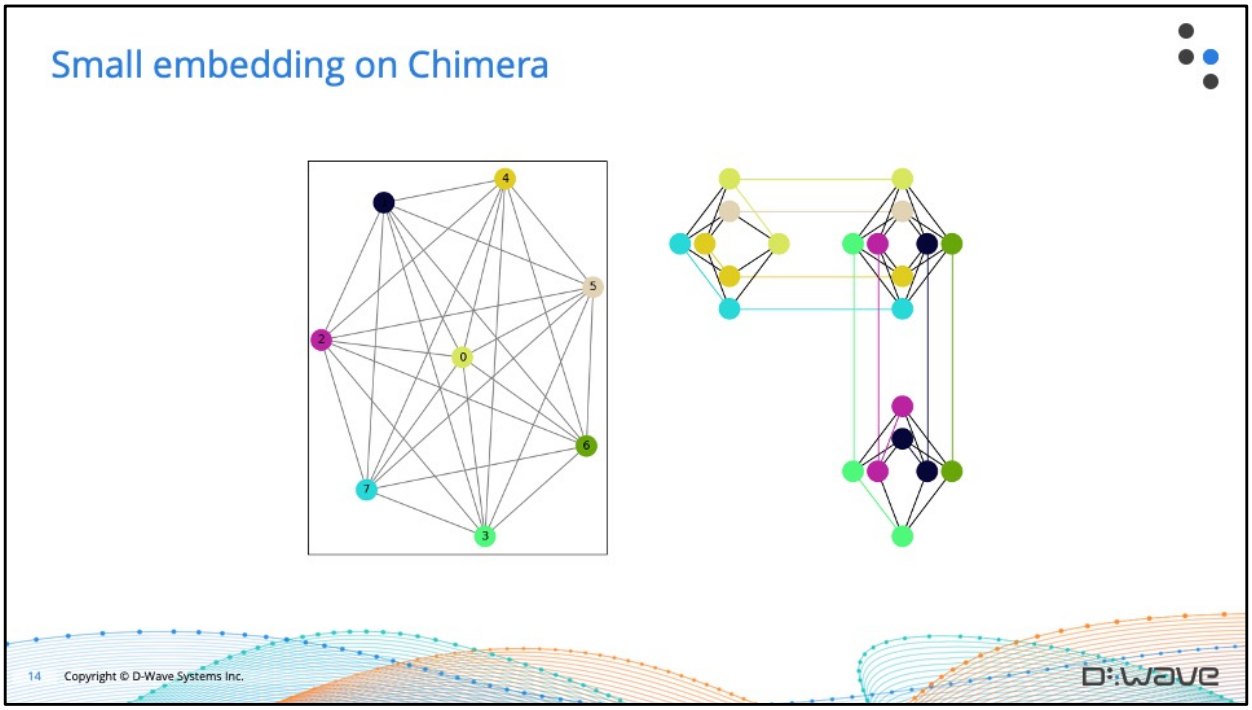
For our 5-variable vertex cover problem, we see that the logical variable or node highlighted in pink in the top picture maps to the chain of two qubits highlighted in pink in the bottom picture. These two qubits together represent our one logical variable. Notice that our logical variable has 3 connections in the top picture. In the bottom picture, our group of two qubits connects to three other qubits. We have our same structure if we visualize that those two qubits are merged together into one.

When the embedding composites in Ocean are used, the appropriate linear and quadratic coefficients, or strengths and biases, are set up on the physical QPU to match your original, logical problem. They will distribute the linear coefficient for the logical variable across the chain that will represent it on the QPU.

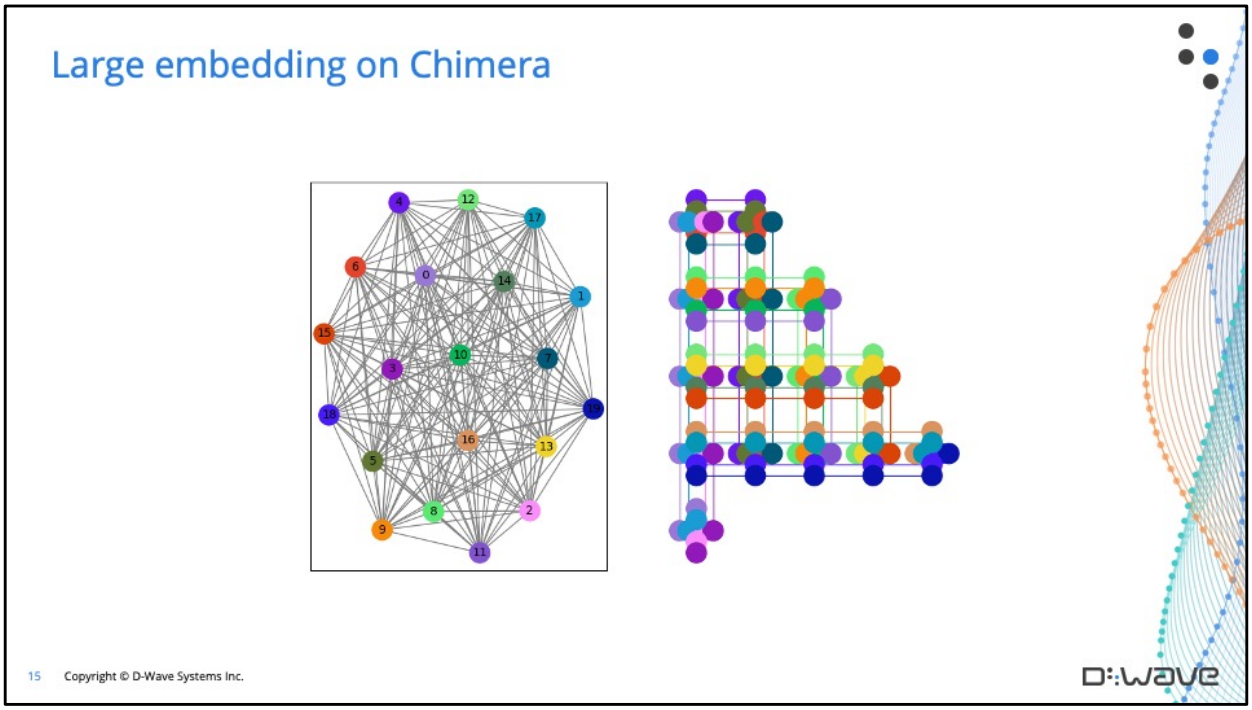


Here's another example. Suppose that our QUBO is represented by this triangle with 3 logical variables. We'll call them 1, 2, and 3. If our physical graph, or our chip topology is a hexagon, how can we map our logical problem to the physical graph? There's a lot of different ways this could be done. We could stretch out one logical variable like variable 1 across 4 physical qubits. Alternatively, we could stretch out each logical variable across 2 qubits. Either of these options (or the many other options you might come up with) are valid embeddings.

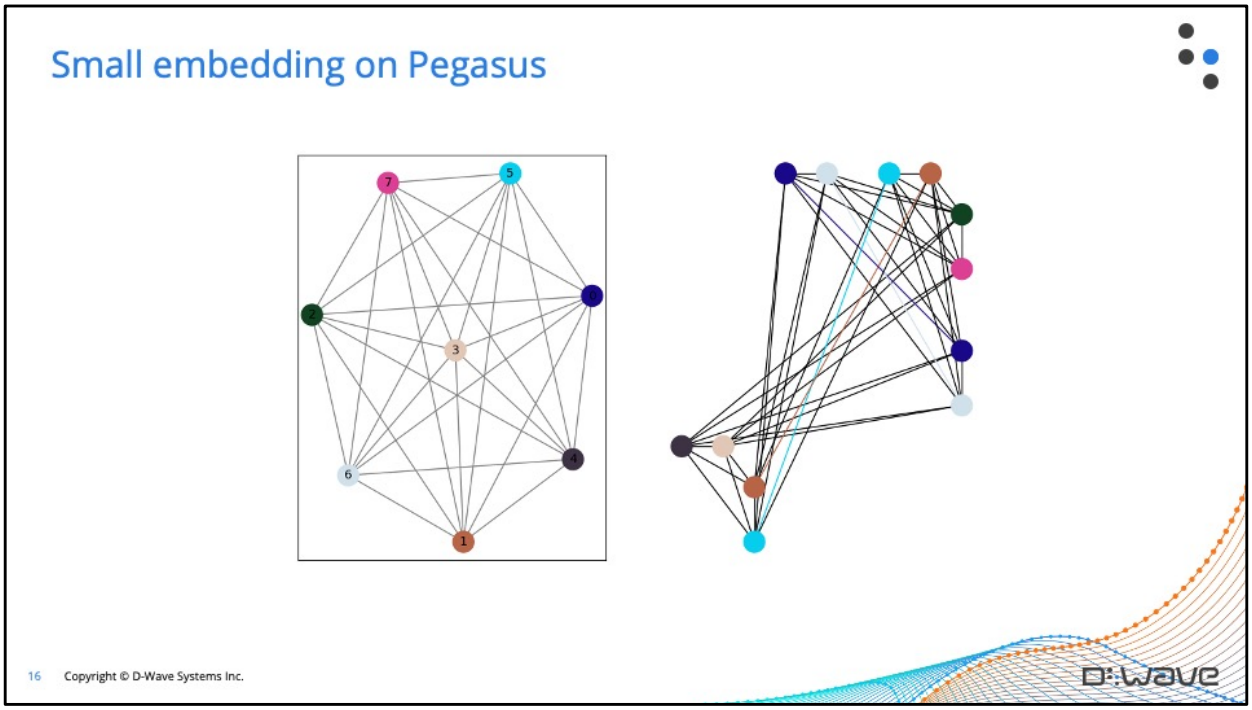
Looking at our hexagon graph in the bottom left picture, think about merging together all of the teal nodes labeled 1. This merging together, or contracting, will result in a triangle that looks like our logical graph in the top picture. Similarly, if we merge together or contract our nodes with the same colors and labels in the bottom right picture, we end up with the same triangle.



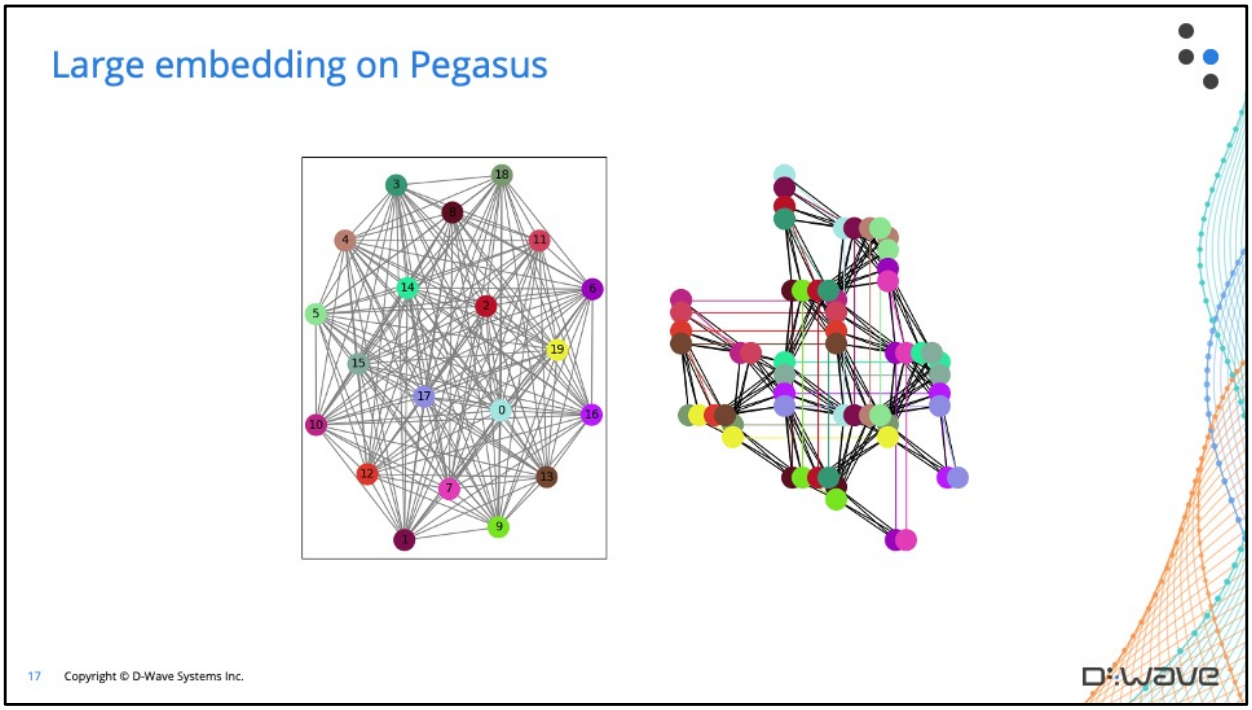
Here's an image of a relatively small 8-variable logical problem and how it might embed onto the Chimera topology with 22 qubits.



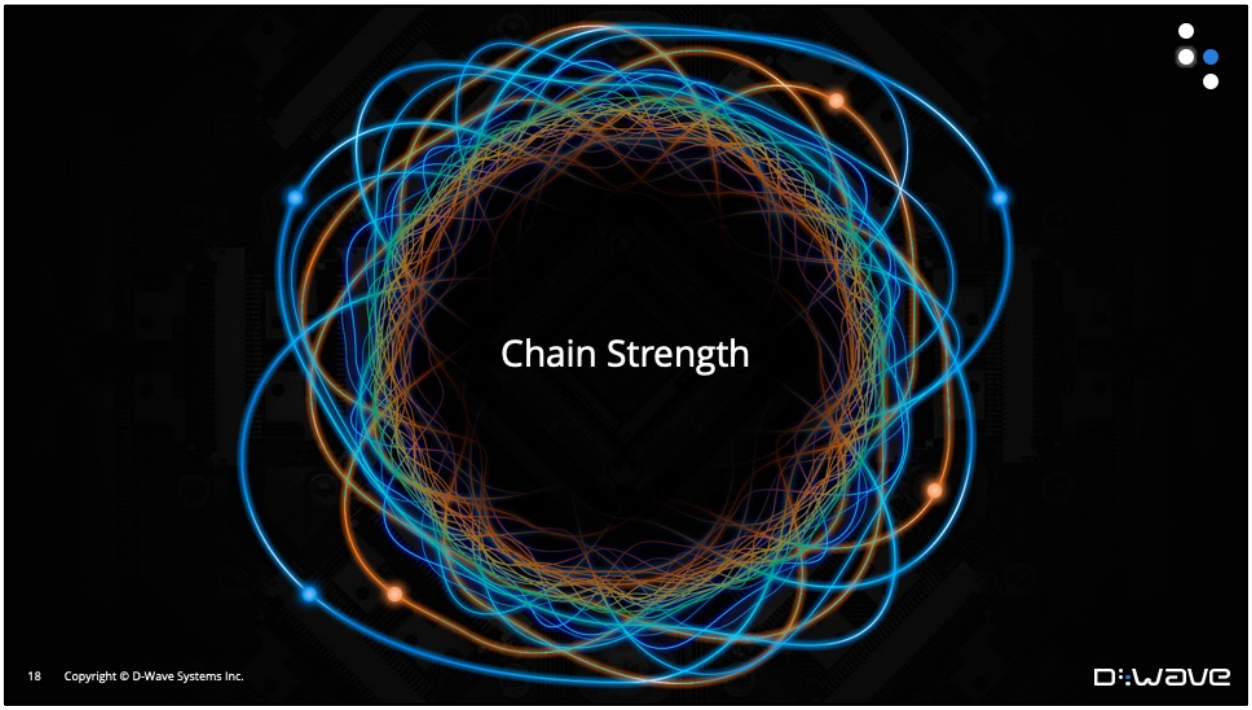
Here we have a larger 20-variable logical problem embedded onto the Chimera topology, which requires 120 qubits.



Now let's look at the small 8-variable logical problem and how it might be embedded onto the Pegasus topology. It uses just 12 qubits, compared to the 22 qubits required on the Chimera topology.

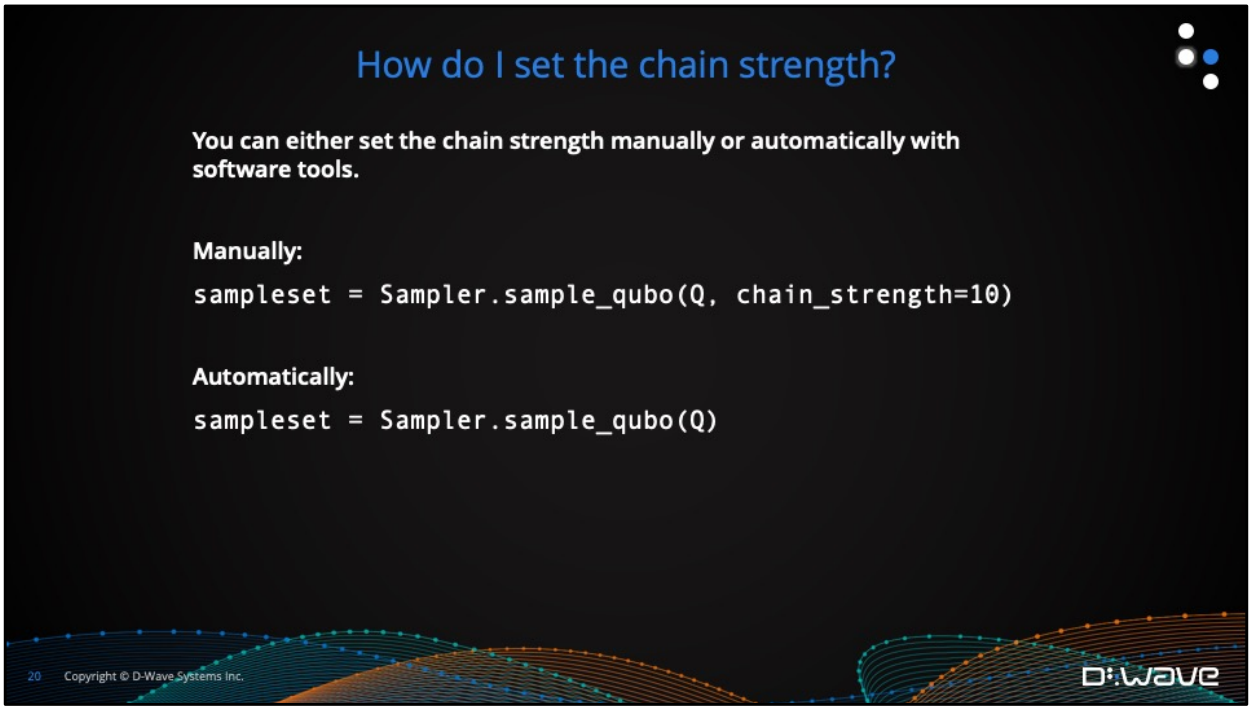


And last but not least let's look at the large 20-variable logical problem on the Pegasus topology. Again, we are using fewer qubits due to the improved connectivity in the Pegasus topology. This embedding required 60 qubits, while Chimera needed 120 qubits.



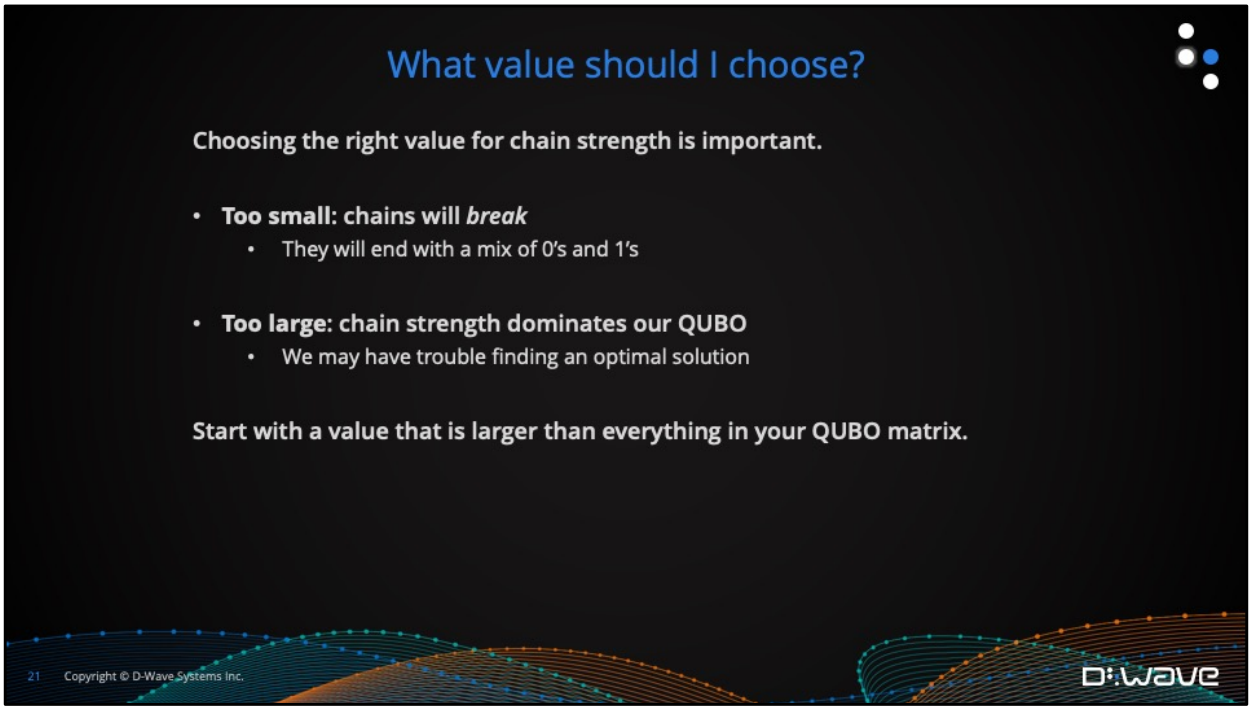
When an embedding has chains, there is a parameter called chain strength that controls how strongly chains of qubits behave as one logical variable.

For qubits to behave together in a chain as one logical variable, we need to have a new constraint in the QUBO problem under consideration. Chain strength is like the lagrange parameter for this constraint. It determines how strongly our constraint that chains behave like one logical variable is enforced compared to the objective and constraints we formulated for our QUBO.



The chain strength value is a number chosen relative to the values in your Q dictionary. It will vary depending on what coefficients appear in the QUBO for the problem. Just like the lagrange parameter, we need to choose the right value for our problem.

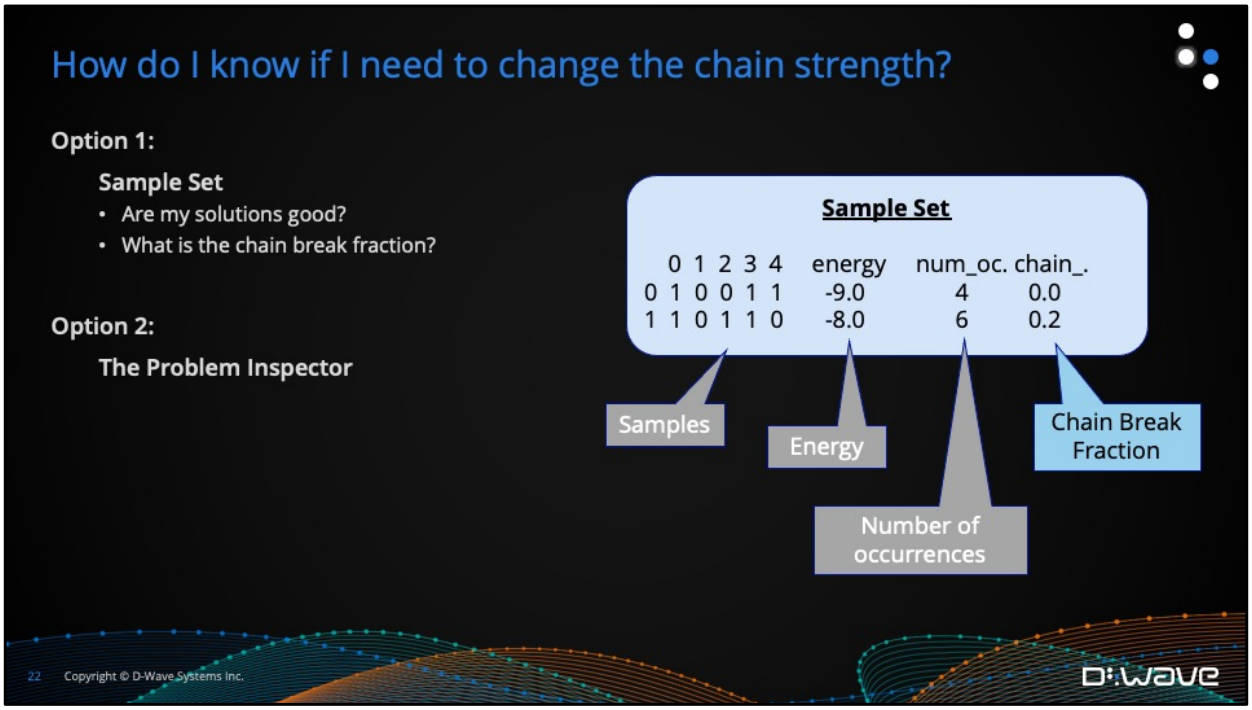
For chain strength, we can either choose this value manually or allow the Ocean software tools to tune this parameter for us. Generally when getting started it's a good idea to let the Ocean tools take care of this parameter. To set this manually we simply add the parameter chain_strength and a value to our sample_qubo call. To set it automatically, simply omit this parameter.



To set the chain strength manually, it's important to choose the right value for your problem. You may need to try a few different values and examine the results returned from the QPU before honing in on the right number.

If the chain strength is too small for a given problem, we will have broken chains. A broken chain is one that returns a mixture of 0's and 1's. Since the chain should behave like one logical variable, a mixture of 0's and 1's makes it difficult to decide what value our variable should be. To resolve broken chains, the software will use a majority vote – if the chain has more 0's then the logical variable will be 0, and if the chain has more 1's then the logical variable will be 1.

On the other hand, if we set the chain strength too large in our QUBO then the chains in our problem will dominate our QUBO. Essentially, the constraint to enforce chains is weighted so heavily that our objective and QUBO constraints become extremely small and insignificant. A good starting point is to choose a value for chain strength that is larger than everything in your QUBO matrix. Try this value out and see what the solutions look like.

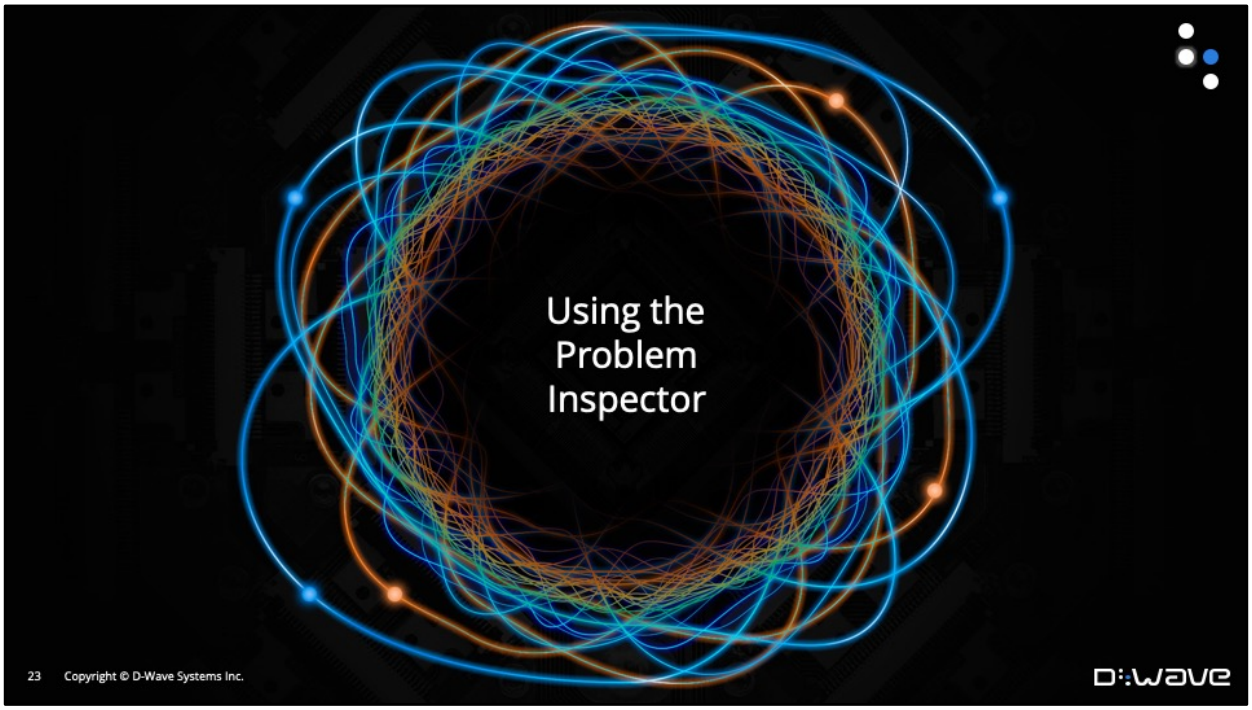


How do we know if we need to adjust the chain strength?

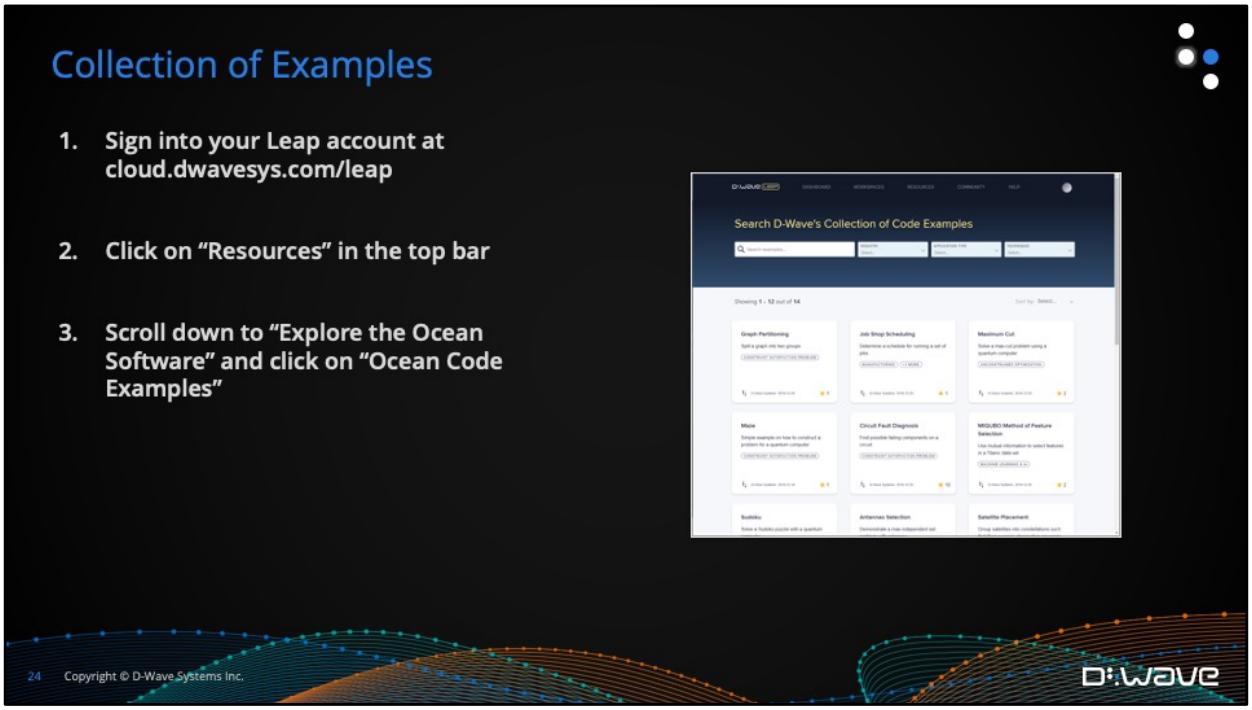
Do you have broken chains? Increase the chain strength. Are there no broken chains but the solutions aren't very good? Decrease the chain strength.

But how do we know if we have broken chains? We can find out using either the sample set object returned from the sample_qubo call or using the problem inspector.

The sample set object provides information on the solutions, or samples, as well as the corresponding energy value, number of occurrences, and chain break fraction. The chain break fraction tells you what fraction of chains in your problem broke. This will be a number between 0 and 1. A value of 0 means no chains in your sample broke. A value of 1 means ALL chains in your sample broke. Using this we can decide if we should increase or decrease the chain strength.

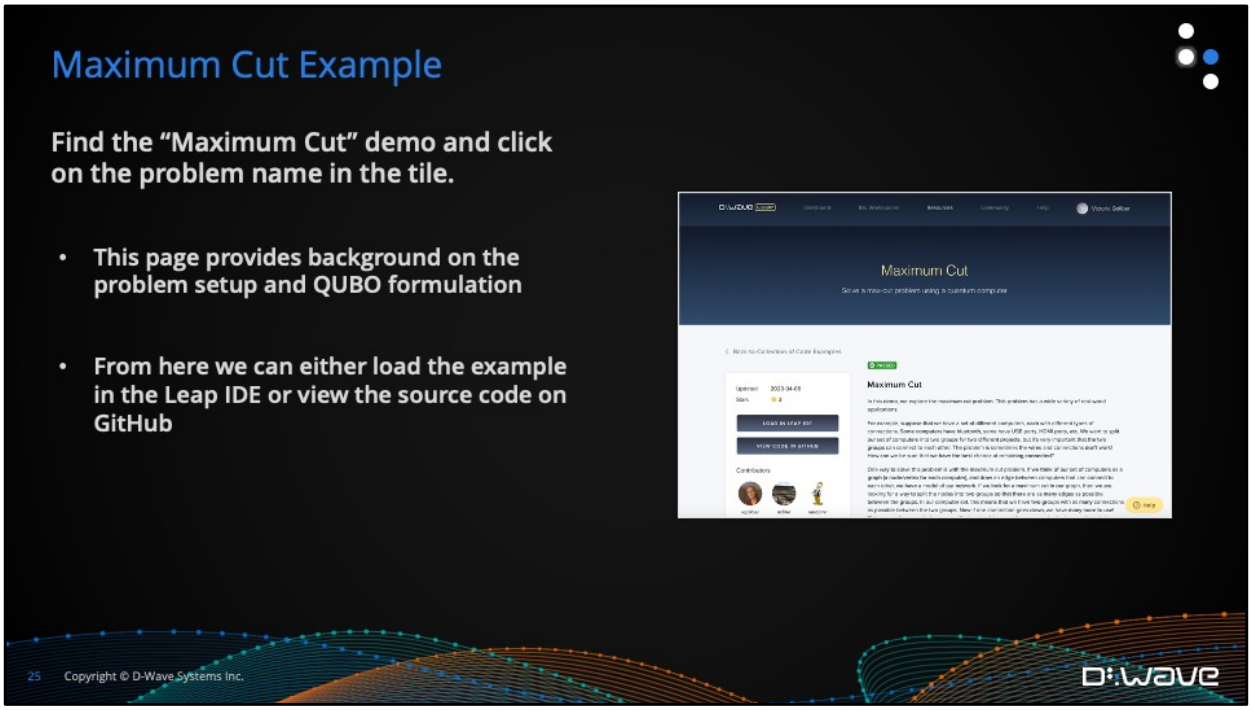


The second method to see if your problem has any chain breaks is to use D-Wave's problem inspector.



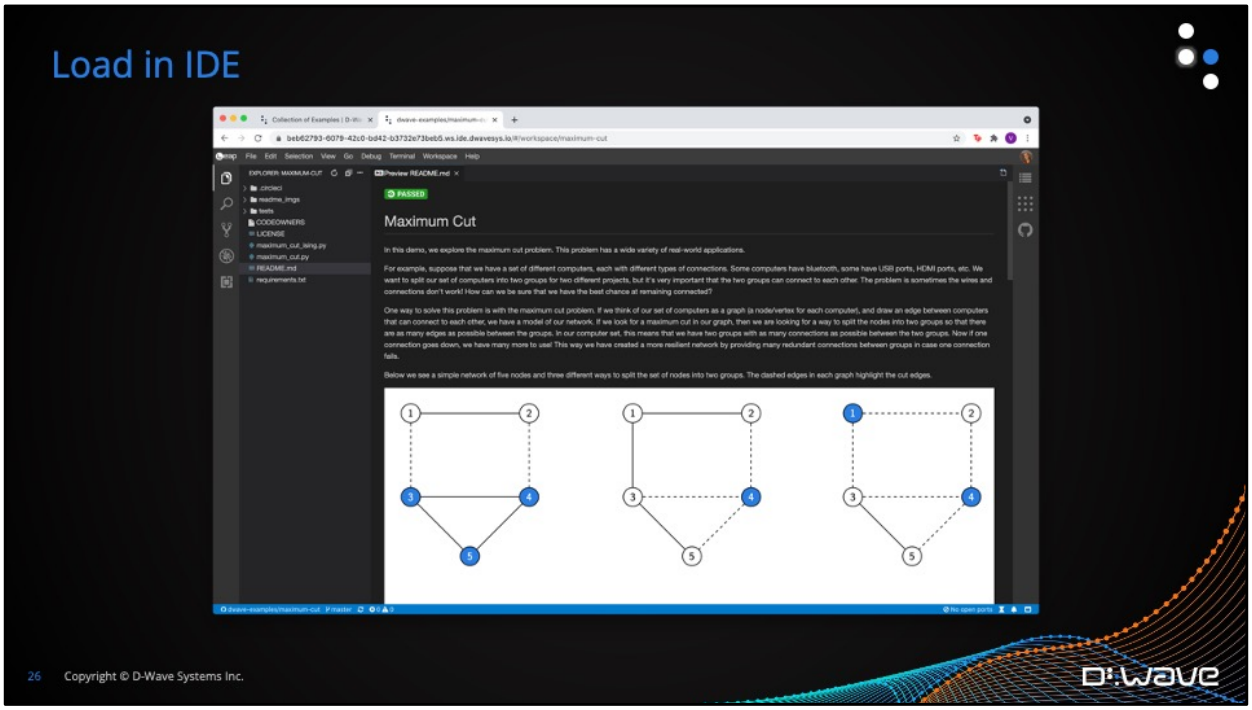
Let’s open up the problem inspector together.

Log in to your Leap account and navigate to the Collection of Code Examples. You can find this through the resources button on the top bar, and then scrolling down to the tile that reads “Ocean Code Examples”.



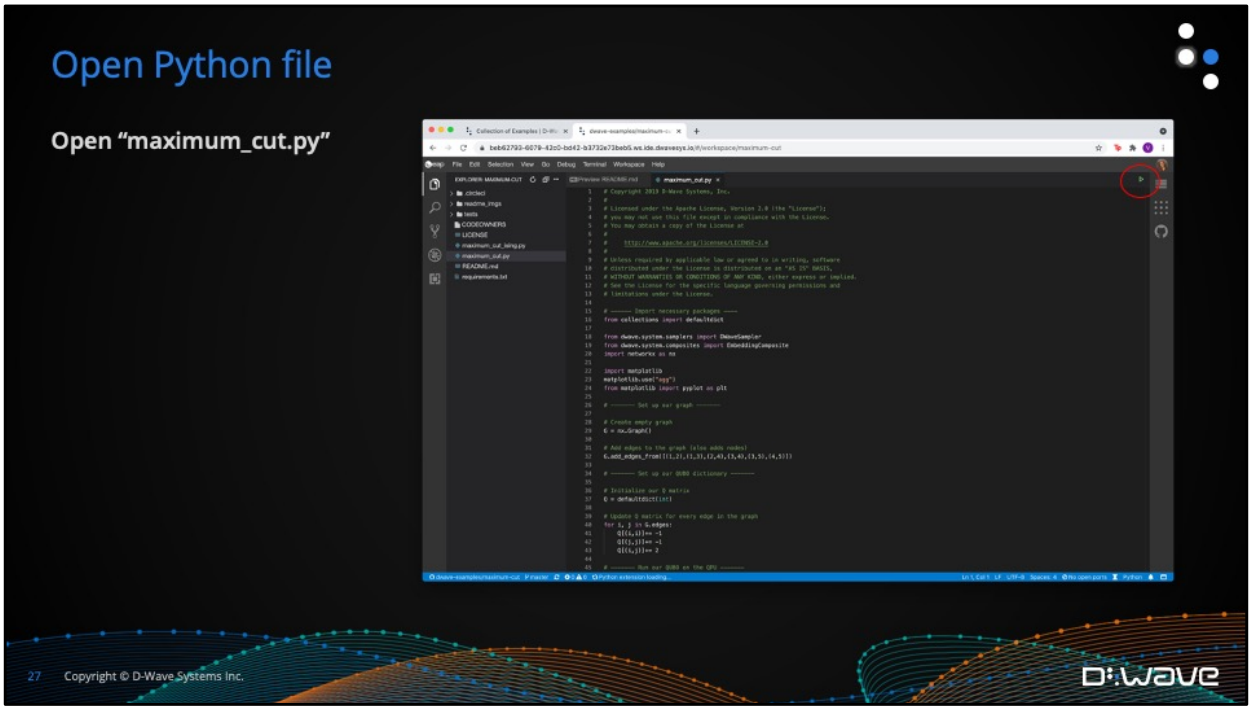
In the Collection of Code Examples, find the maximum cut demo and click on the problem name. You should see a description of the problem and how it's formulated, as well as instructions on how to run the code.

On the left side bar there are buttons to load the example in the Leap IDE or view the code on GitHub. Click on the blue "Load in Leap IDE" button and start up the IDE.



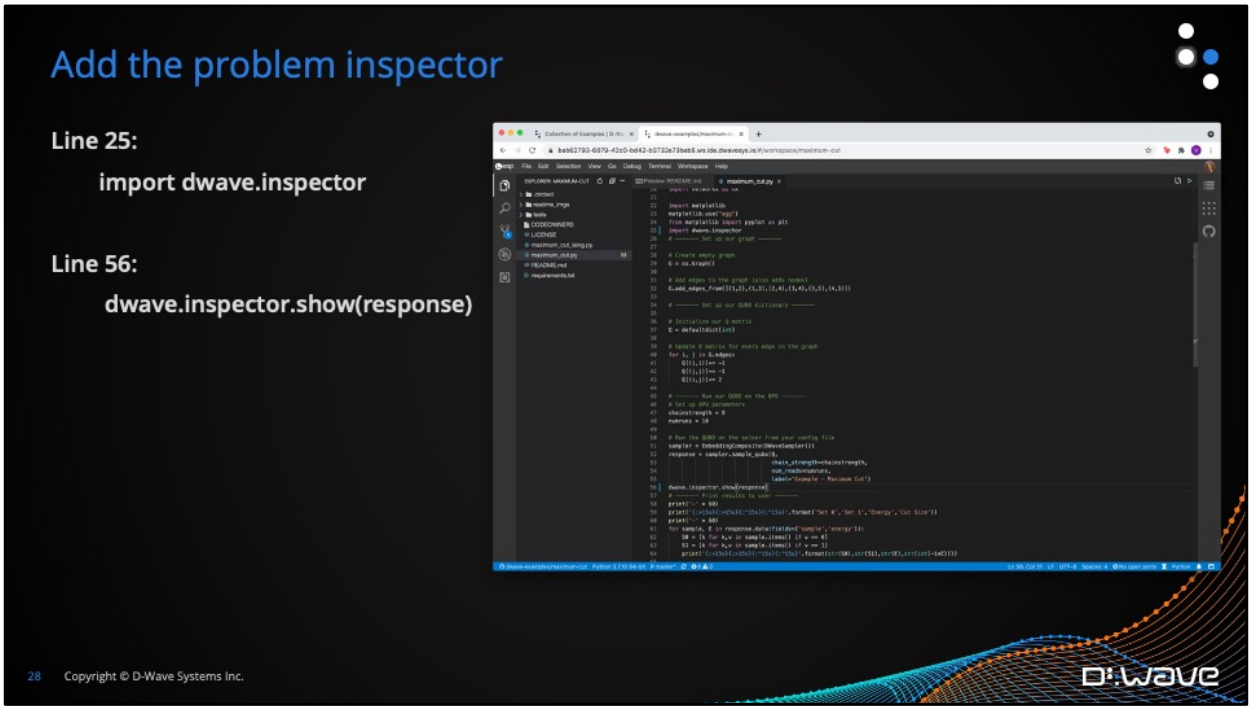
The workspace will take a few seconds to load. It’s building a virtual environment for the demo to run in and linking to your Leap account and solvers.

Once the workspace has loaded, you’ll see the problem description, or README, on the screen, along with a file browser on the left sidebar.



Double-click on the file “maximum_cut.py” in the left sidebar to see the python code to run the QUBO problem.

In the top right is a green triangle – clicking on this will run the code example on the QPU. Give it a try to make sure everything is working as it should be.

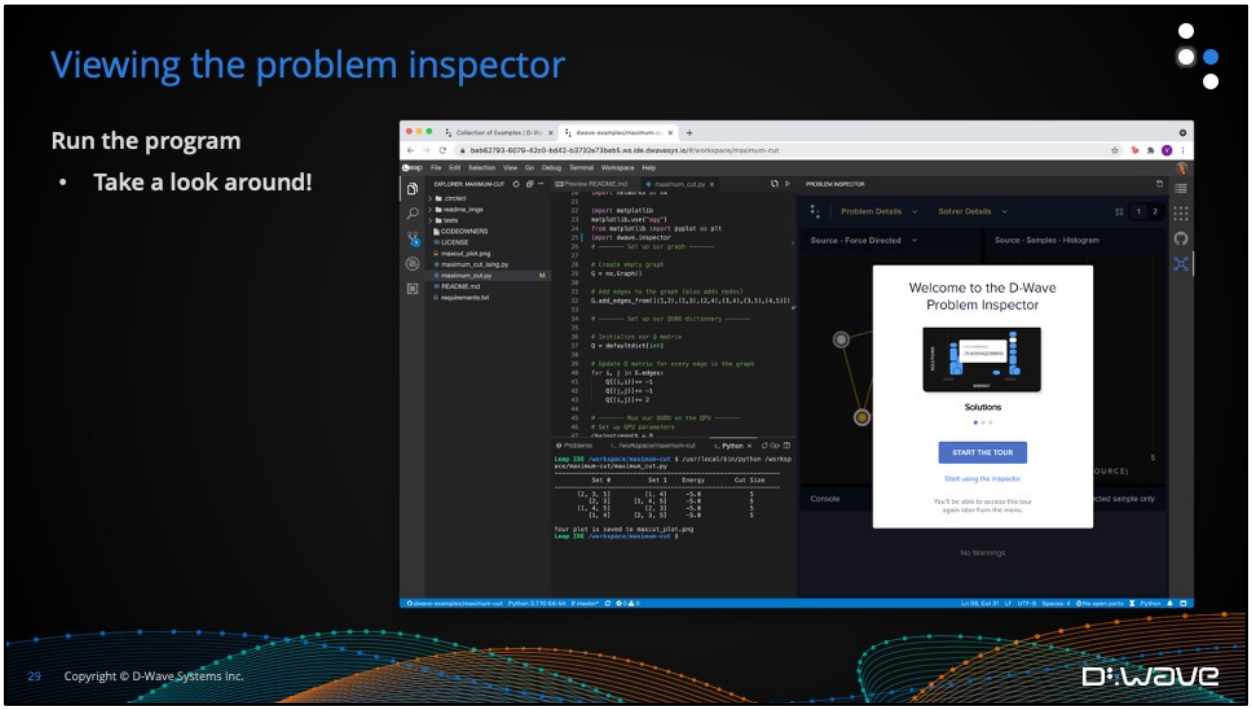


Let’s add the problem inspector to our program.

First we need to import the package that contains the problem inspector. Near the top of the python file add “import dwave.inspector”. This needs to be done at the top of the file so that the inspector can collect information when you send your problem to the QPU.

Next we call the problem inspector to show us our solution. In this program when we run sample_qubo we store the sampleset in a variable called response. Just beneath our sample_qubo call add “dwave.inspector.show(response)”.

Save and run the program again.



When you run the program the problem inspector should open automatically on the right side of the web browser. The first time it runs there's a click through tour that you can take to see all of the different features. Close the tour by clicking on "Start Using the Inspector".

Within the problem inspector there's a lot of different information.

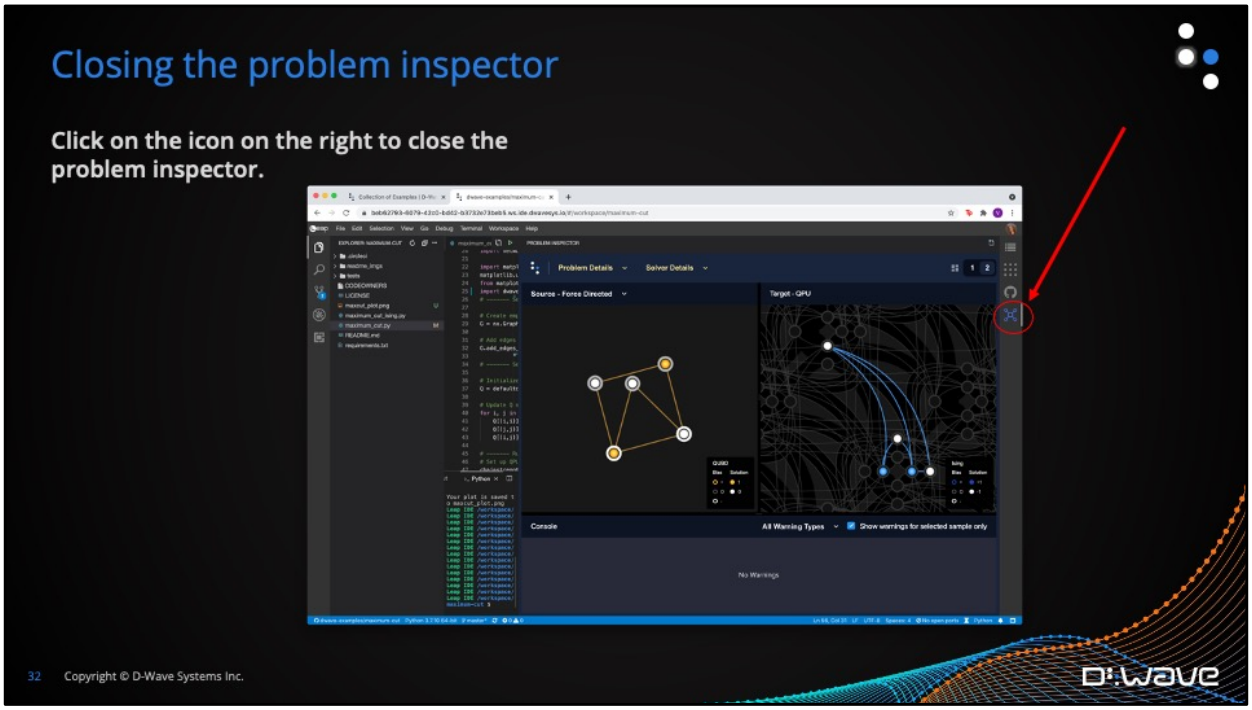
On the left we see our logical problem. Each circle or node in this picture represents one variable in our QUBO problem, and each edge or line connecting two circles represents a quadratic interaction between two variables.

The picture on the right shows a histogram of the different samples returned from the sample_qubo call. Think of this as a bar graph where the height of the blue bar shows how many samples had a certain energy value. Remember that lower energy is generally best, and so the samples to the left side of the picture are usually the best solutions found.

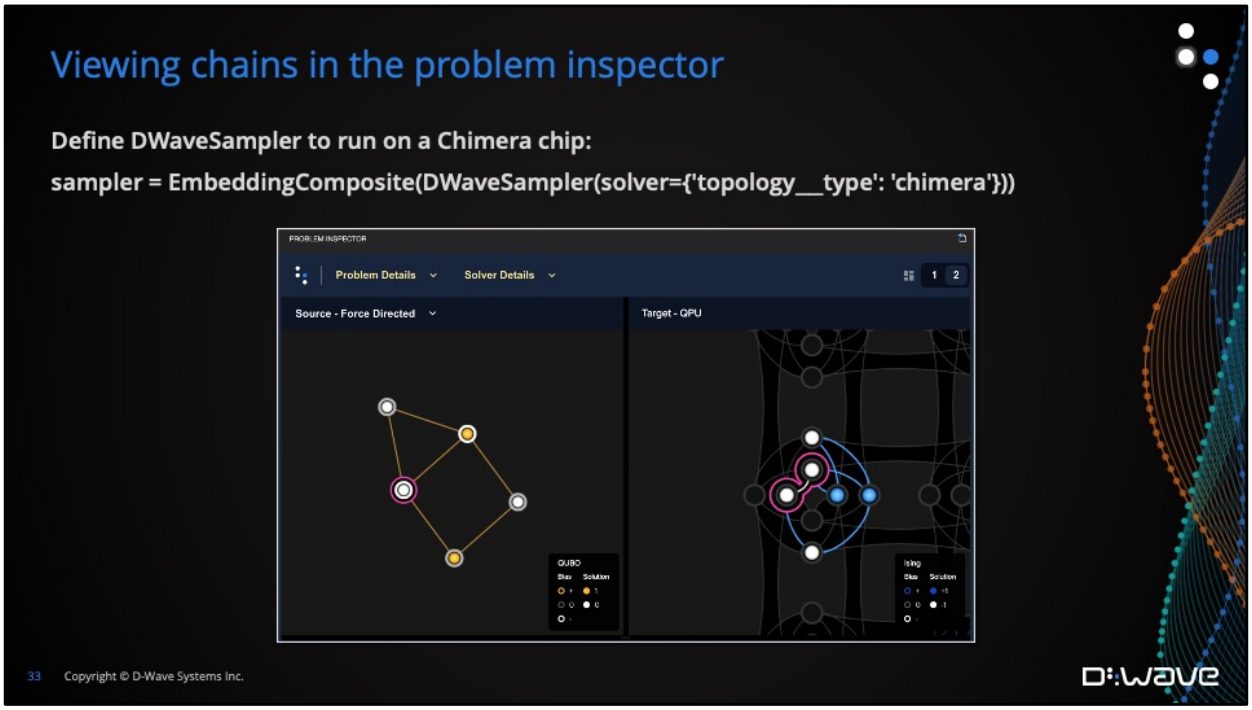
At the bottom of the inspector is a warning box. Here's where you might see some warnings about chain strength values being set too low for your problem.

At the top right of the inspector are numbers 1 and 2. We are currently in view 1. Click on the 2 to switch to view 2.

In view 2 we see the same logical problem on the left and warning box at the bottom, but the image on the right has changed. On the right we now see how our logical problem was embedded on the physical QPU when it ran. Yours might look a bit different if it ran on a different QPU or with a different embedding.



To close the problem inspector, click the icon on the right that looks like an X.



When I ran this problem in the previous example there weren't any chains used in the embedding. This specific logical problem doesn't need any chains when embedded on a Pegasus topology, but does need chains on the Chimera topology. Let's force the problem to run on a Chimera chip to see what chains look like in the problem inspector.

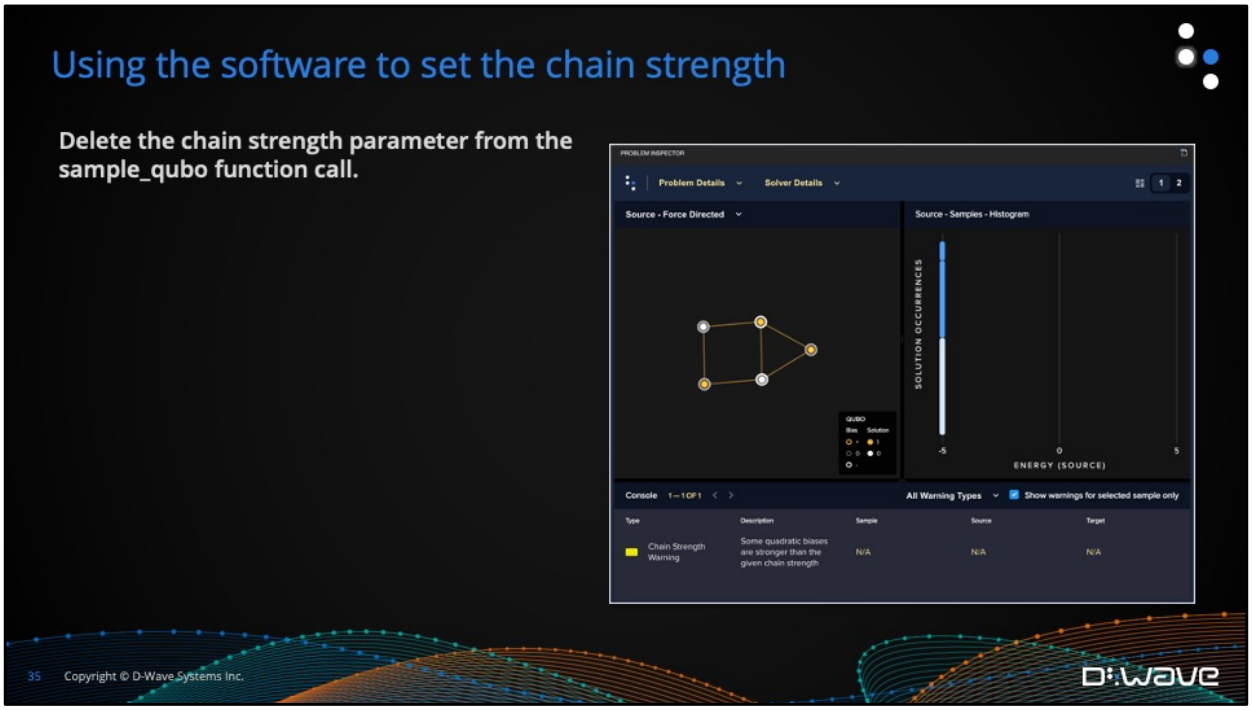
We have defined our sampler as the DWaveSampler using the EmbeddingComposite. The EmbeddingComposite is mapping our problem to a QPU. To force the sampler to use a Chimera QPU, we add a parameter to the DWaveSampler call on this line. Inside the pair of parentheses immediately following DWaveSampler, add in solver={'topology__type':'chimera'} to run our problem on a Chimera QPU. Note that the underscore in this command is a double underscore.

Save and run the problem, and look at view 2. Clicking on each logical variable should show you the associated qubit or chain in the embedded problem. Can you find a chain in the embedding?

The problem inspector can help us to identify if our samples contain broken chains.

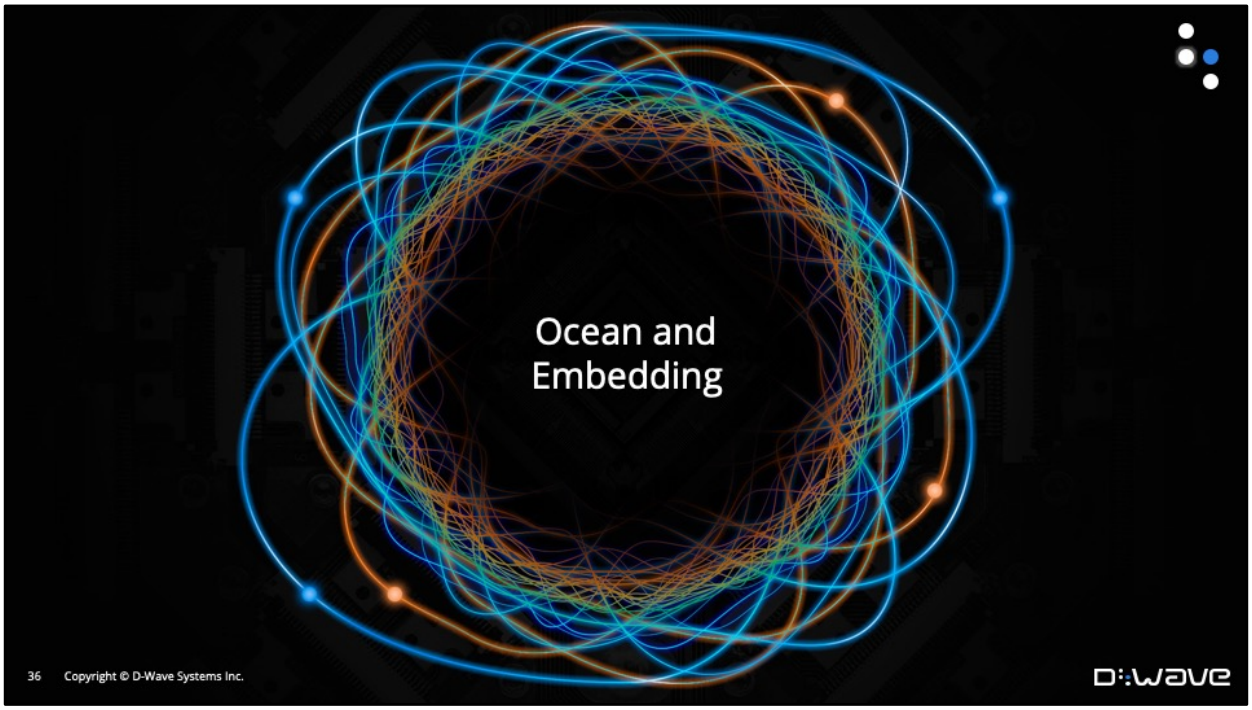
To see this, let's adjust the chain strength in the program to a small number. Let's choose a really small number like 0.5. Now save and run the problem and take a look at the problem inspector.

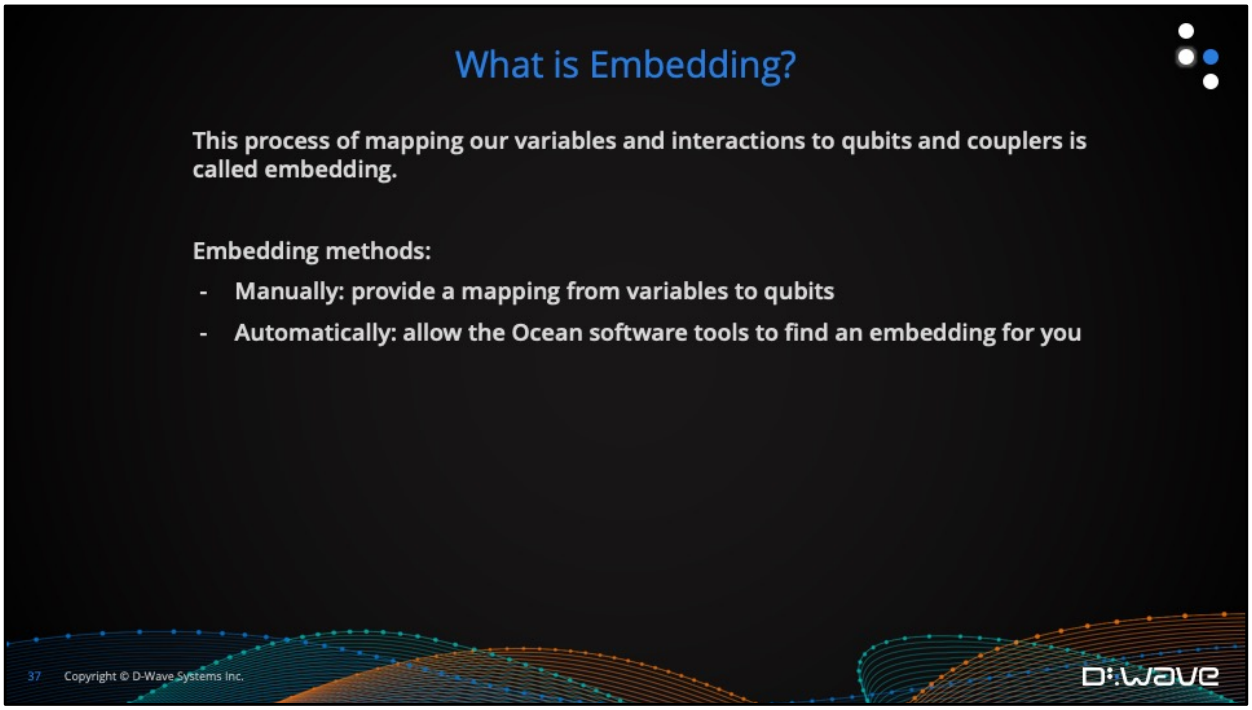
Clicking on each sample in the histogram on the right we can see different samples that were returned. Each sample that we click on colors the nodes in the logical problem differently to show the solution that was returned. If a node is both yellow and white like this one, it means the logical variable was represented by a chain that is broken in that sample. We also see this by looking at the warnings in the warning box at the bottom of the window.



In this program the chain strength is set manually. To use the software tools to automatically set the chain strength, delete the chain strength parameter in the sample_qubo call. By deleting this parameter we are letting the software tune the chain strength for us.

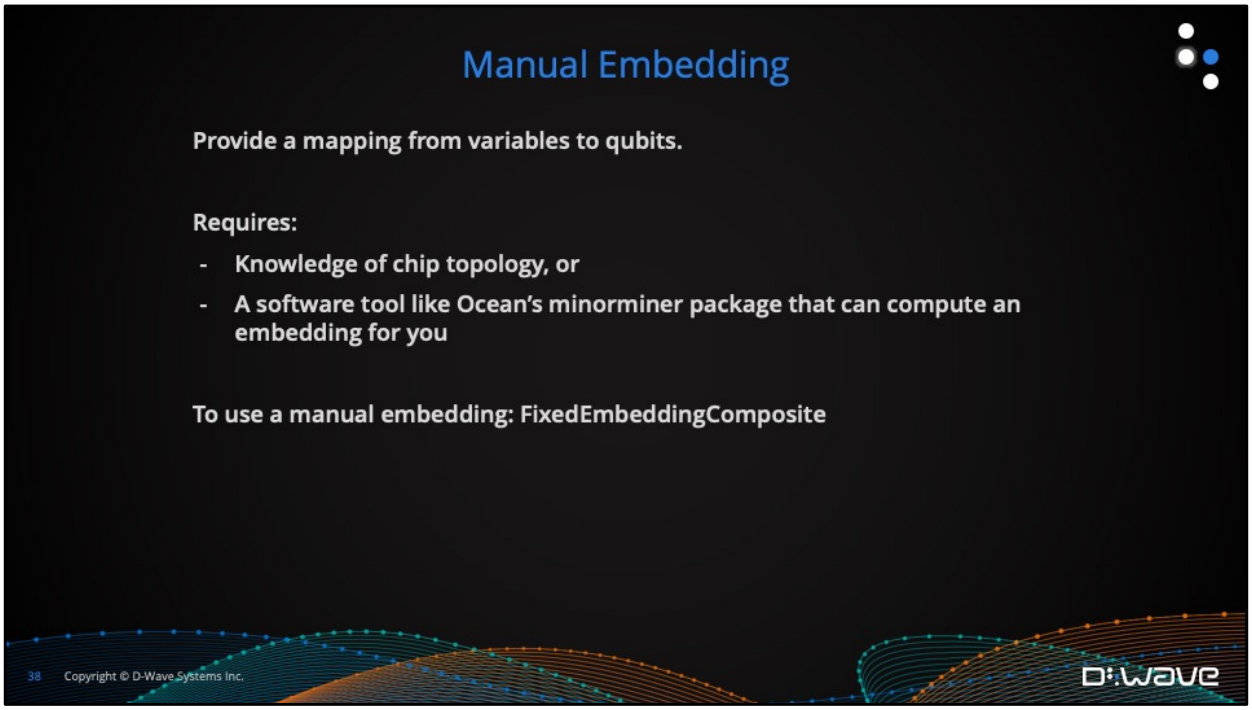
Save and run the program again and see if there are any chain breaks in your samples.


Ocean and
Embedding
36
Copyright © D-Wave Systems Inc.
D:wave



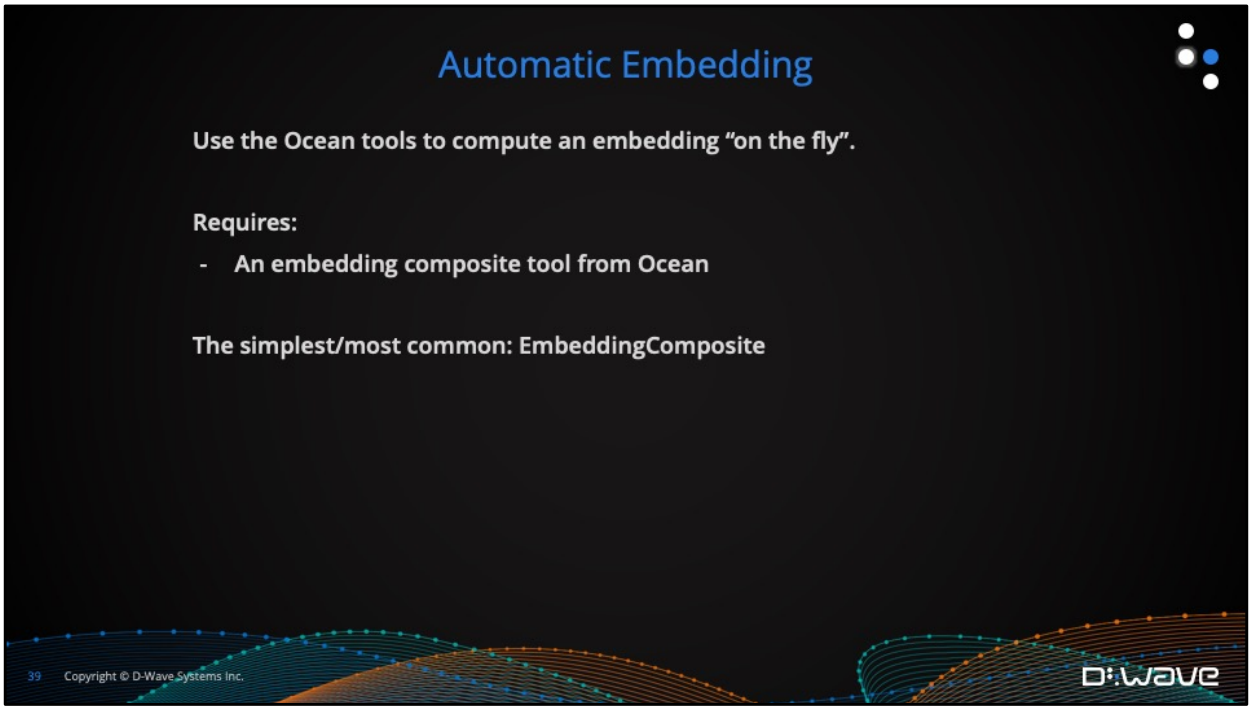
Every problem that we run on any quantum computer needs to be embedded to match the topology of the QPU. Luckily the Ocean SDK provides a few different methods for embedding your problem.

There are two ways that we can embed problems on D-Wave's QPUs. We can either embed them manually by finding an embedding that matches our problem's QUBO to the chip topology, or we can use the Ocean software tools to automatically embed our logical problem for us.



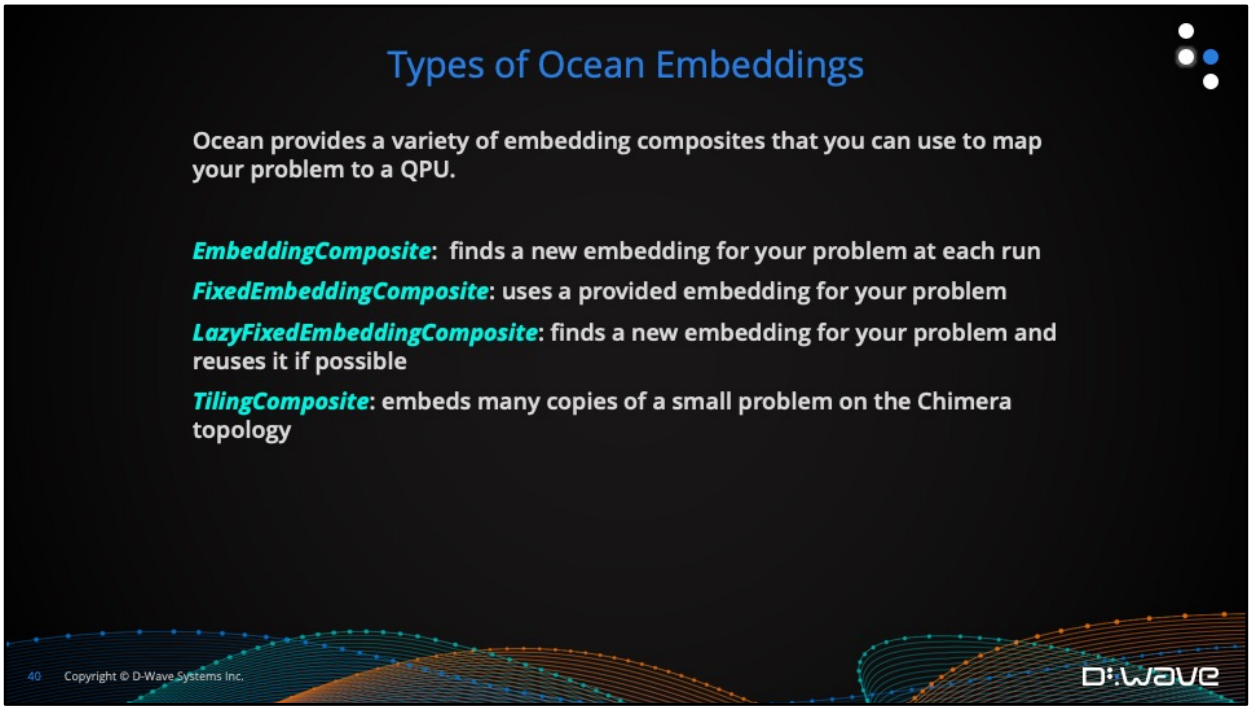
To use a manual embedding, we must come up with a mapping of our logical variables to the physical qubits. For example, we need a mapping that says something like "variable x1 will be represented by qubit 42". For small problems we might be able to do this by hand, but for large problems the mapping quickly becomes very complicated to calculate.

You can use an embedding package like minorminer to compute an embedding from your logical problem to the target QPU. Then, we can run our problem on that QPU using Ocean's FixedEmbeddingComposite.

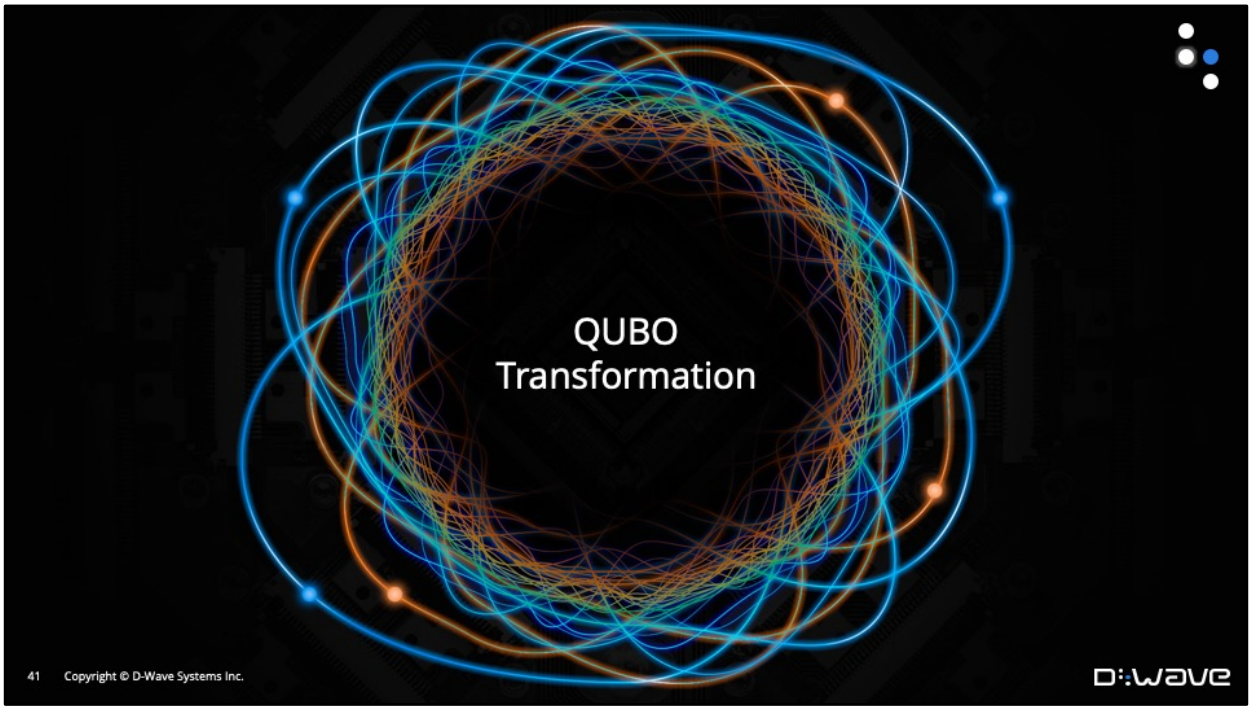


A simpler method is to compute an embedding "on the fly" while running your problem on the QPU. This will combine the function call to sample your QUBO with an embedding function. In one line in your Python program you can both embed and sample your QUBO on the quantum computer!

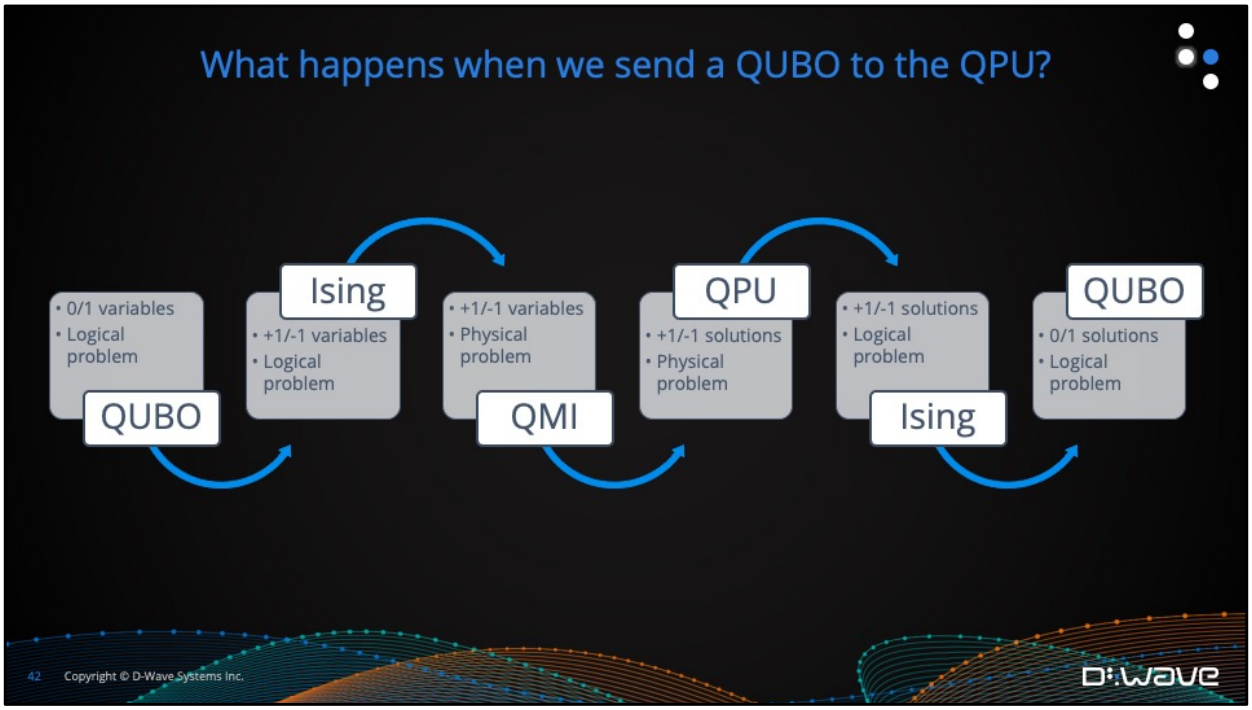
The simplest (and most common) way to do this is with Ocean's EmbeddingComposite.



We've seen two different types of embedding tools or composites that Ocean provides already. There are a few more available in Ocean as you explore more types of programs and applications. Here are a few that you might want to take a look at.

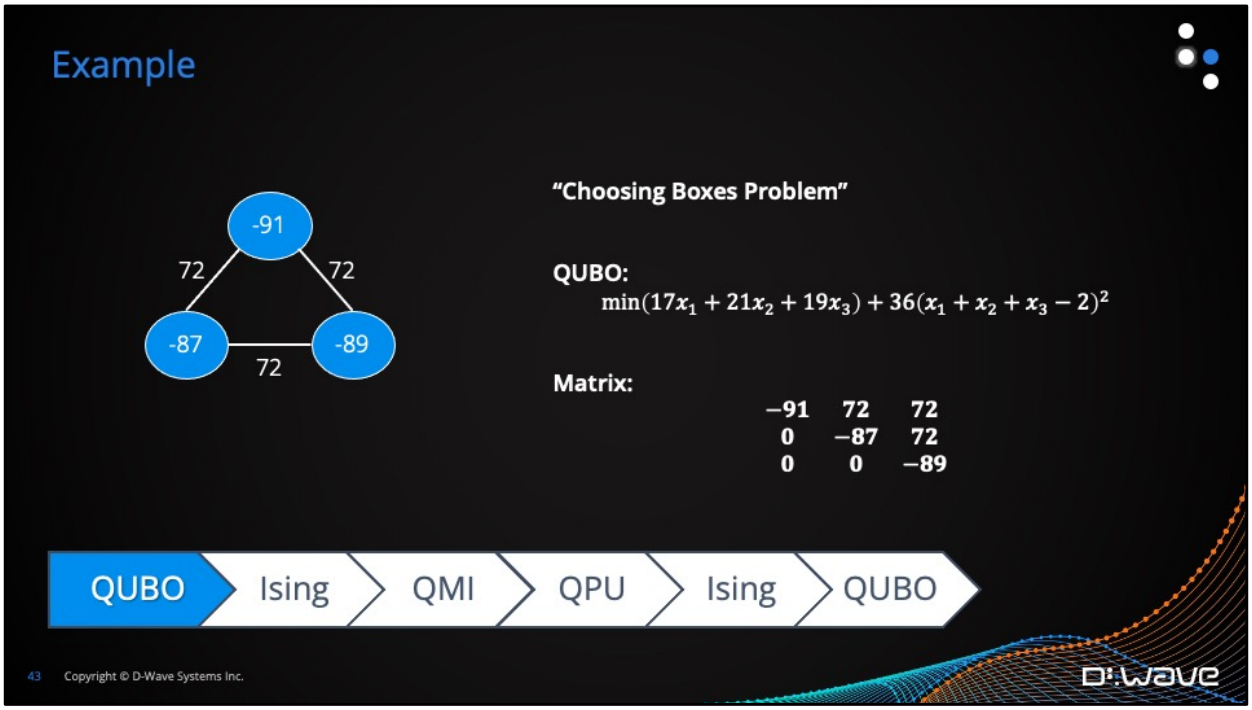


Now that we know how the logical problem structure is transformed when it is embedded on the QPU, what happens to the coefficients and biases in our logical problem?

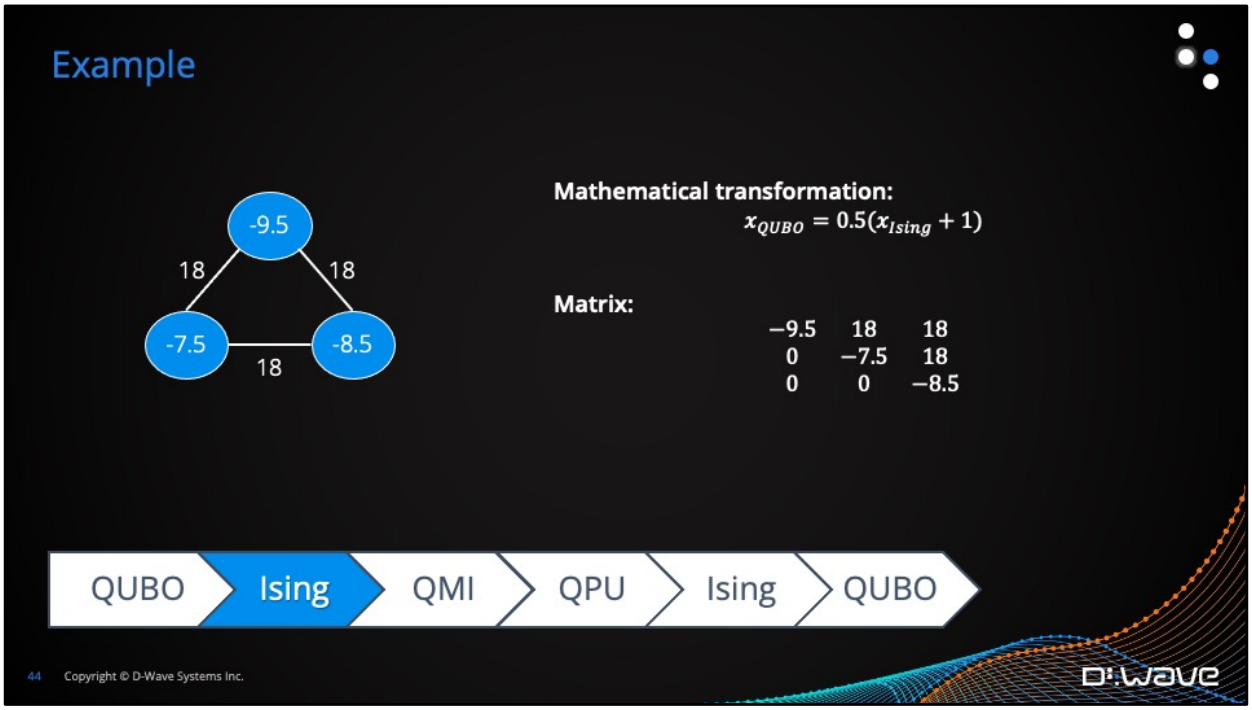


When we send a QUBO problem over to the QPU, it undergoes a transformation along with the embedding process.

First, our QUBO is converted to an equivalent Ising model, which is then embedded to create a QMI. This embedding adjusts the problem biases to the ranges available on the physical QPU and assigns biases to chains. This QMI is run on the QPU, and solutions to the Ising model are returned, which the software converts back to our original 0/1 binary QUBO solutions.

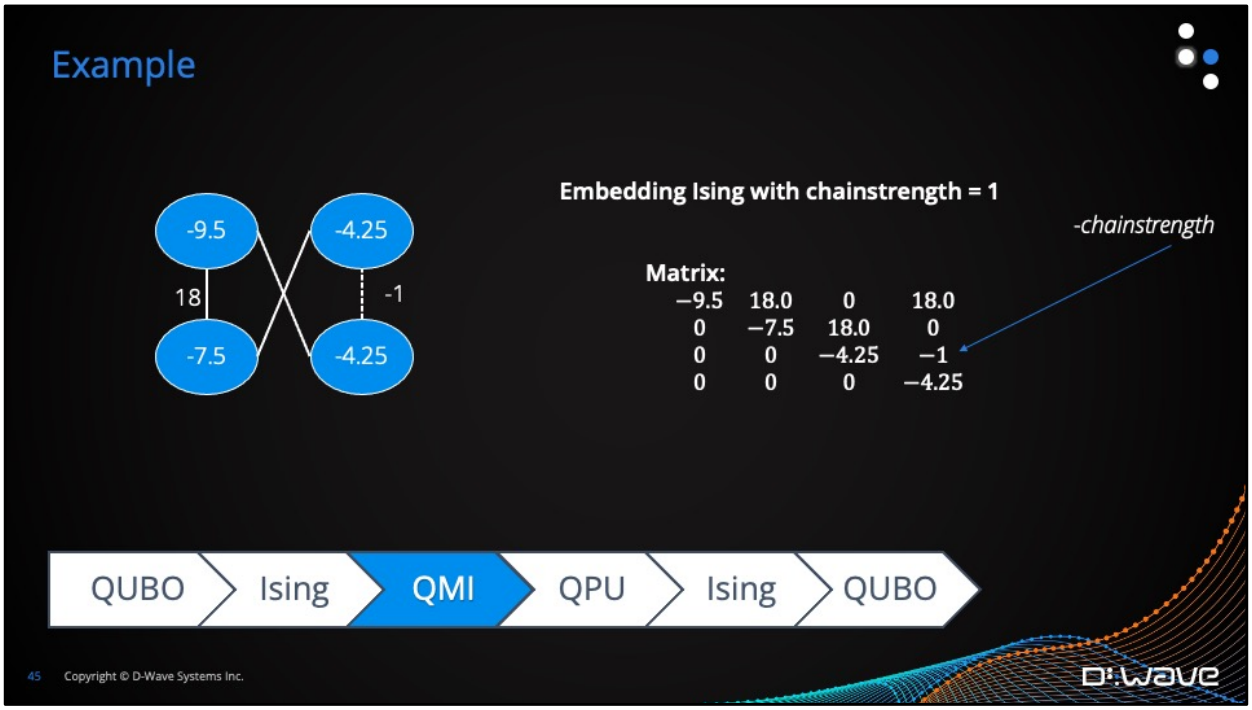


In an earlier module we developed a QUBO for the choosing boxes problem. In this problem we have three boxes with values 17, 21, and 19. We must choose the two boxes with the smallest sum. Here you see the QUBO that we developed, and we're using a Lagrange parameter of 36. Viewing this QUBO in the matrix form, we see the linear coefficients along the diagonal and the quadratic coefficients in the upper triangular section of the matrix.



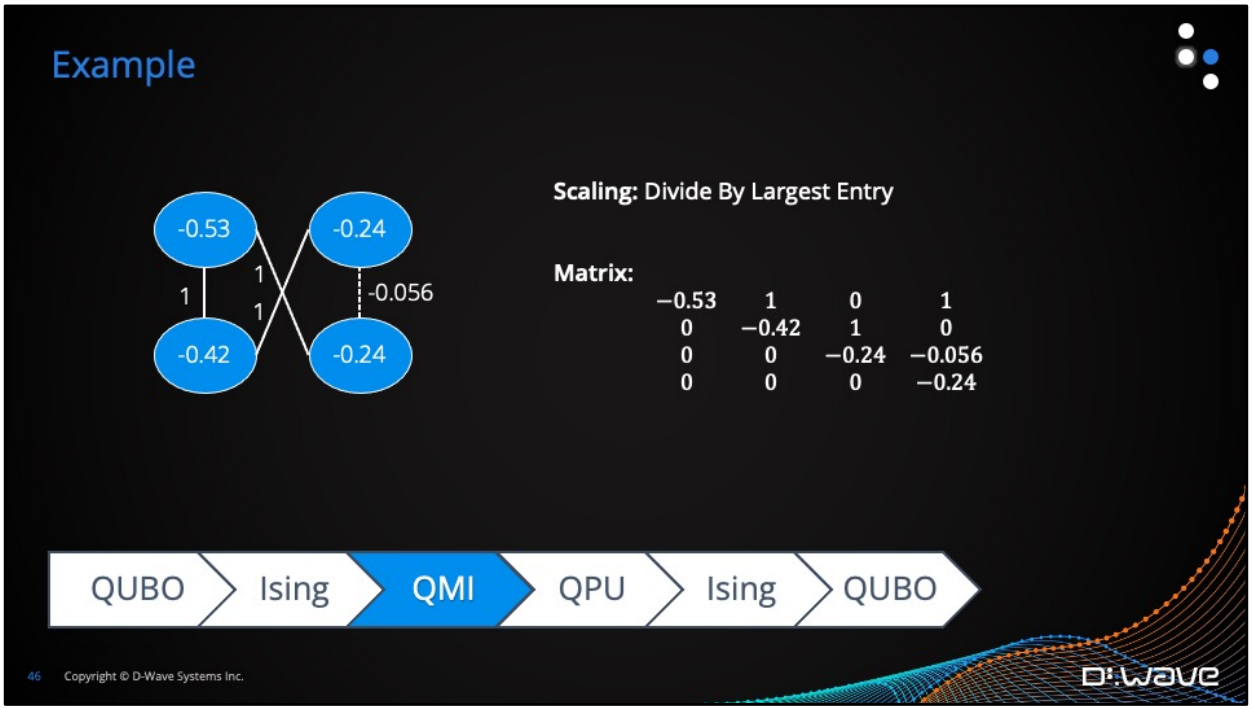
We have been working with the QUBO model where our variables can take the values 0 and 1. An equivalent form is the Ising form where the variables can take the values -1 or +1. There's a simple mathematical formula to convert from QUBO to Ising, which is shown here.

D-Wave's hardware natively implements the Ising form, so our QUBO is first converted to an Ising problem by the software tools.



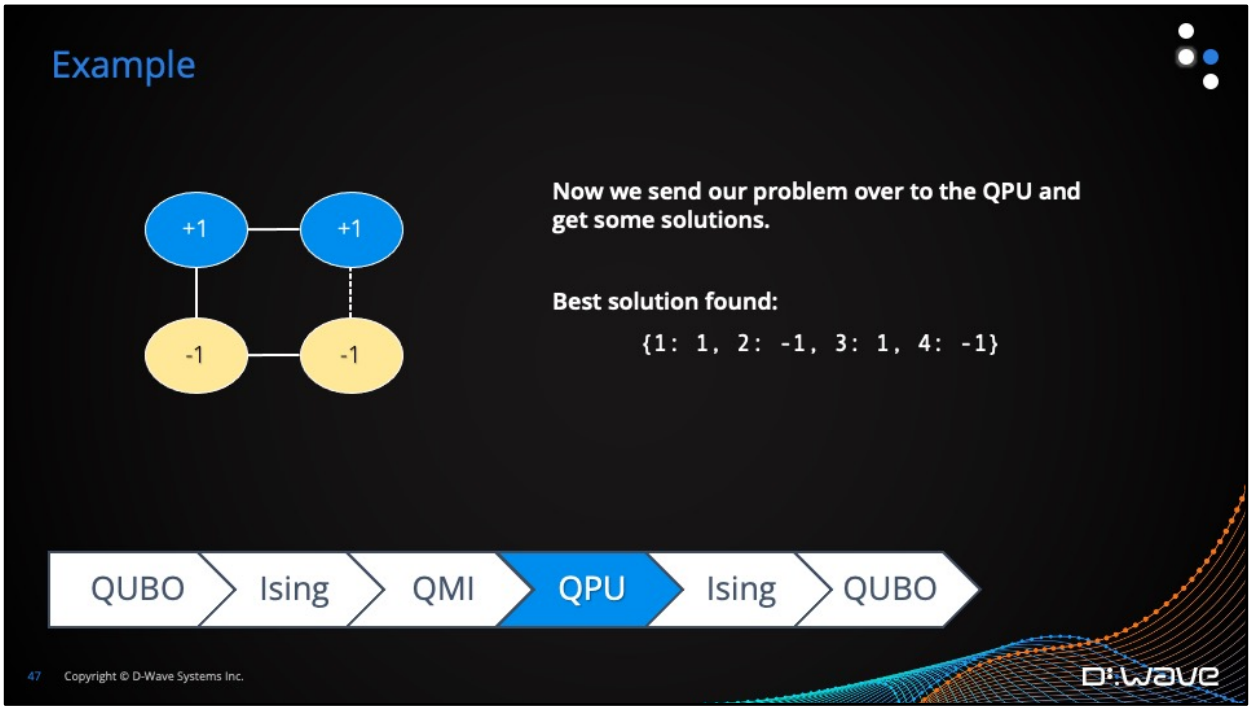
Once the Ising form for our problem has been created, the problem is embedded onto the hardware. Here is where chains are created, and the chainstrength provided in our original problem is used to assign a bias to the edges in the chain.

Our Ising model had a linear coefficient of -8.5 on one logical variable. This logical variable is now split across a chain of two qubits, each of which has half of the linear bias. The edge connecting them is assigned a quadratic bias of -1 times the chainstrength value, and the remaining biases remain the same.

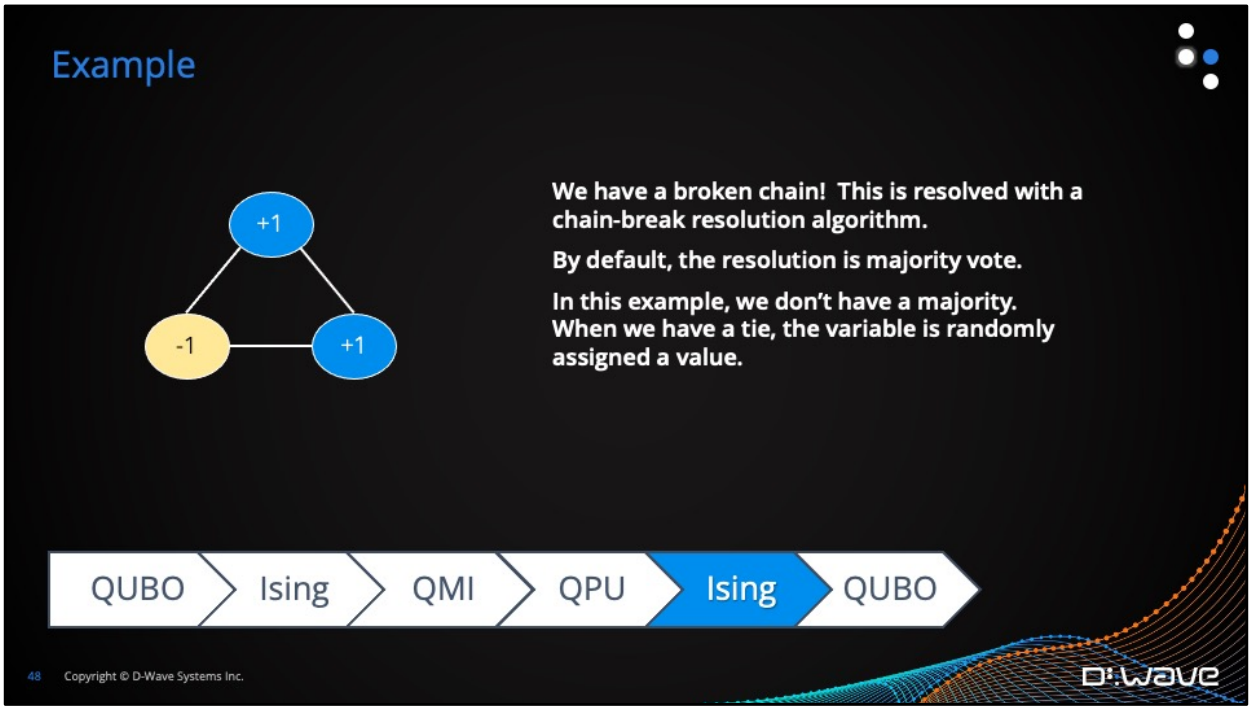


The hardware allows for quadratic biases in the range of -1 to +1 and linear biases in the range of -2 to +2. To get all of our matrix entries down to these ranges, we divide by 18 (the largest number in our matrix). This is called scaling.

Our QMI is now complete and ready to run on the QPU.

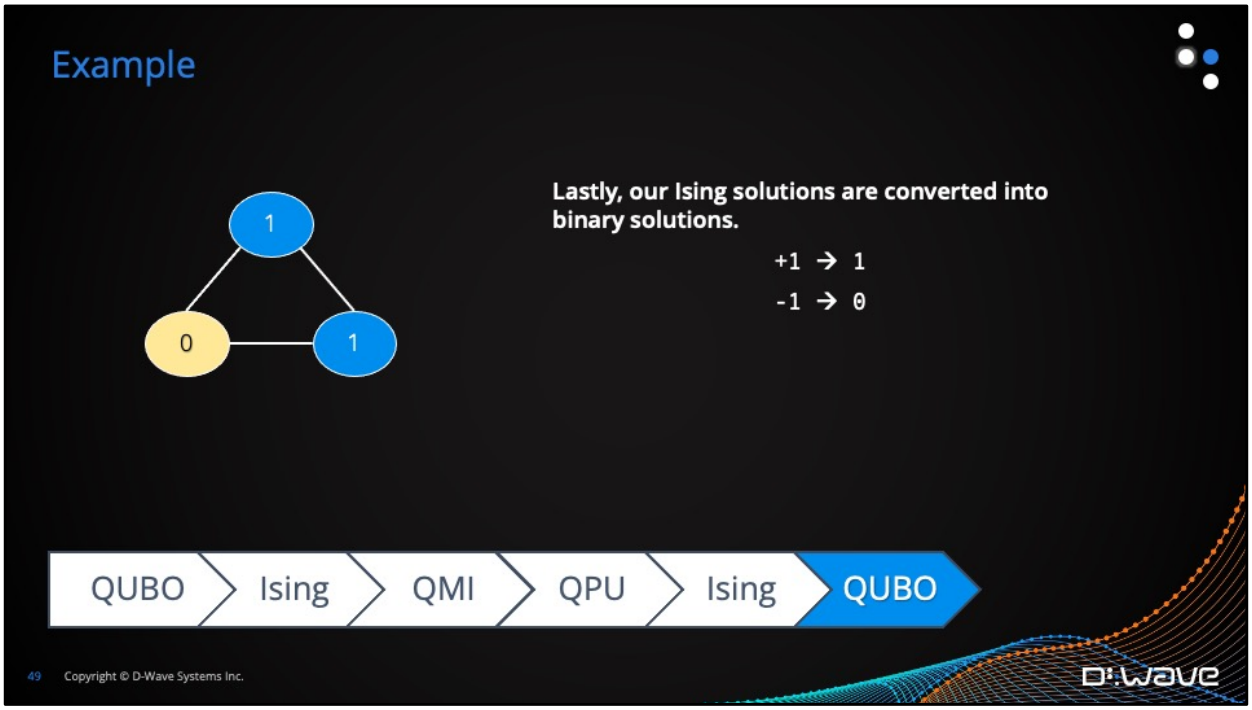


When our problem runs on the QPU, we get a collection of samples – one for each anneal or read that we asked for. These solutions are for the embedding Ising model. In this example, we might get a solution back that looks like this. Note that our chain of two qubits has one qubit with a value of -1 and the other with a value of +1, and so we have a broken chain.

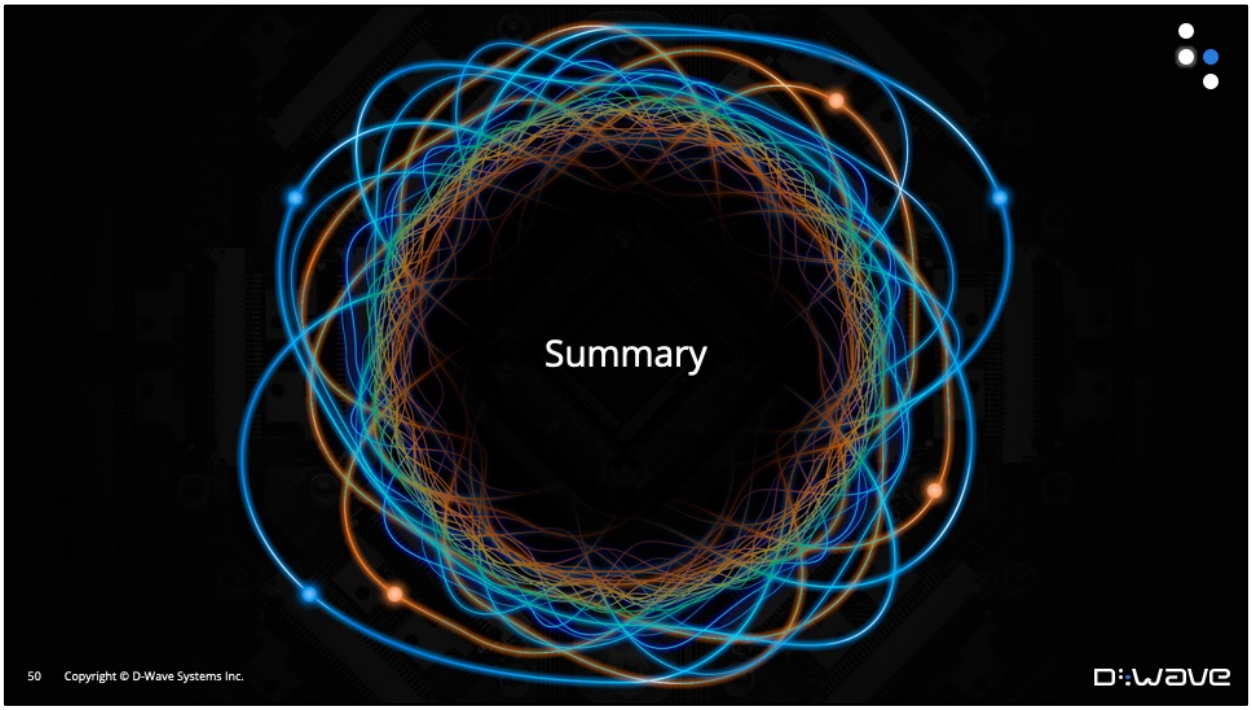


Since our chain represents one logical variable, ideally our qubits in the chain should all return with the same value. If they don't, then we need to resolve our broken chain and choose a value for our logical variable. In general, the software uses a majority vote. There are other options available that you can play with as well.

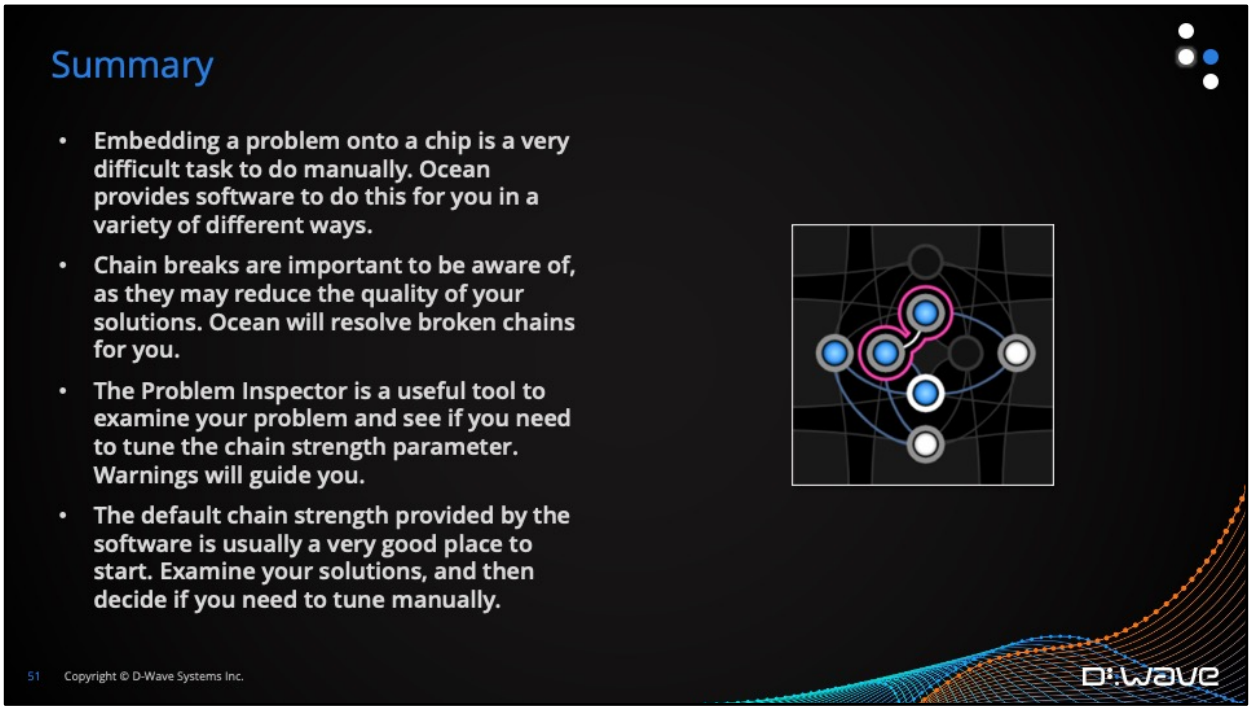
In this case, we don't have a majority since there are only two qubits. This means that the software will randomly choose a value for our logical variable that is represented by this chain.



Last but not least, the Ocean tools convert our Ising solution to a QUBO solution by mapping each +1 to 1 and each -1 to 0. These are the solutions returned to you in your sampleset.



As you have seen, there is a lot that takes place behind the scenes when you send a QUBO or BQM over to D-Wave's QPU.



As we've seen, embedding a problem onto a QPU chip is a difficult problem and the problem biases undergo quite a transformation. It's not very easy to do manually, but our software tools have algorithms built-in to take care of this for you. Depending on your problem, there are a few different embedding composites that you might want to explore.

When embeddings use chains, we have an additional tunable parameter we need to be aware of: chain strength. As we start learning about Ocean programs, we can let the software tools take care of this parameter for us. However, depending on the quality of the solutions returned you may need to manually tune the chain strength for your problem.