

[SQUEAKING] [RUSTLING] [CLICKING]

SPEAKER: OK, today is the last of the formal lectures, and I'm very pleased that Samuel Bruce is here to present. Sam and I have been working together, and hopefully, you'll see this is the culmination and synthesis of bringing computer science and economics together on the same page. Thanks, Sam.

**SAMUEL
BRUCE:** Thank you. Yeah, I'm Sam. So I'm currently a graduate student at MIT in computer science and been working with Professor Townsend for the last year or so and really trying to take the things that we've learned in this class and other computer science and economics concepts and bring them together into practical applications that can be deployed as good examples for infrastructure to be built in the future. So today, we're going to talk about some of the things that we've been looking into-- specifically market mechanisms and alternative equilibria. So we're going to be using algorithmic game theory and coordinated equilibria, which we'll mention, to find computationally feasible algorithms for mechanisms we can put in real code.

So just a roadmap of what we're going to go through. We're going to start with Dubey's limit order market mechanism, which is a very specific mechanism that we're trying to implement on the blockchain in code. We're going to talk about the complexity of computing Nash equilibria and how it relates to that market mechanism.

We're going to talk about correlated and coarse correlated equilibria, which are alternate equilibria to Nash that might be more tractable, easy to compute. We're going to talk about why it matters, what the constraints on blockchain algorithms are, and tractability concerns that arise when trying to find equilibria of mechanisms. And finally, we're going to talk about specific machine learning algorithms, which can be used to improve this process, make it more efficient. So we're going to jump right into Dubey's limit order mechanism. And this is a specific game from Dubey from a while ago, basically detailing a limit order market that in equilibria has very nice properties.

And just the notation of the market here-- we have n players notated by i and k goods. Each player has an endowment. We'll denote that with a and a utility function. And in each time period, a player gets to submit a strategy. And what the strategy looks like is these four quantities here-- p , q , p tilde, and q tilde.

And what those denote-- denotes is I will buy q units of good j if the price is p or less. Or conversely, I will sell q units if the price is p tilde or more. So in one strategy vector, each player i is denoting what they would do as a limit order over all n goods-- or all k goods, sorry. And they're allowed to both say that they will buy and sell all goods, even the same ones, at whatever price they so choose.

So a lot of flexibility in the strategy space. They're allowed to submit a bid of these four vectors. And they can choose any combination of what they would like to buy, sell, and what prices. The only constraint on their strategy is that they may not sell goods that are not in their endowment. They cannot go over their endowment set for selling in each time period.

So the full strategy space is the combination of all of these player strategies. We'll denote that as S here. And in outcome consists of x where each x is just the bundle of goods that each agent buys or sells in the time period-- so the trades in the limit order that are actually executed each time period. We denote the entire game as this gamma function here, where E is our market, as we've described it above, and also this weight λ , where λ is a penalty to players who at the end of the period have negative credit.

So notice when I said the constraints on the strategy functions, there was no constraint that a player cannot buy goods he does not have the money for. If a player is endowed with goods and money, they may sell goods and buy goods as long as they don't sell goods they don't have. And at the end of the period, if they after their trade settle, have negative credit or the goods that they buy outweigh the goods they sell plus their endowed money, they incur a penalty. So it's allowed, but they are penalized weighted by λ .

So the actual mechanism implementation for each of the goods we're going to establish a separate trading post. So each good essentially has its own market where the specific orders for each good will filter. The players submit their own strategy in each time period. And for each trading post, all of the bids that the players submit are accrued, and you match the highest buyer with the lowest seller.

So intuitively right-- if the highest buyer is less than the lowest seller, no trade will occur because the seller won't trade for a price that low, and the buyer won't trade for a price that high. But if there is some intersection where some buyer is willing to pay and some seller is willing to sell for less, that trade will execute at the buyer's amount. And it'll match up. So if the first buyer only wants to buy two units, but the first seller wants to sell 4, the first buyer will buy those two units, and then the mechanism will move to the second-highest buyer to execute the rest of the seller's order and vice versa. If the seller wants to sell less, then it will execute in full to that first buyer, and then we'll move on to the next seller.

So as I mentioned on the last slide, each player can borrow at no interest to fund their purchases. So in each period, they can make whatever trades they'd like. However, if at the end of the period they have a net negative credit between the goods they purchased and the goods they sold and their endowed money, they incur a weighted cost by λ based off of the amount of negative credit they have.

So here we're going to define the equilibria of the mechanism. So we have our strategy set. And this is a specific strategy for one time period, little s . So all the players submit their bids. We have a payoff function, which is just the utility functions for each player, capital π_i . And we denote the mechanism to be efficient if there is no subset of players, subset m , who can play an alternate strategy and find a Pareto-dominant outcome for that time period.

So here, we have our payout function across all players. If the payout of the strategy given E -- so E is the deviation here. So this is the payout with deviation is greater than the payout without deviation for all players. And for some player, the payout with deviation is greater. It's a Pareto-dominating bundle, and that is not the equilibrium in this set. It's not efficient here. That's how we define it.

So we can have a coalition of arbitrary size all the way up to every subset of the n players. We call a non-cooperative equilibrium of the mechanism if it is efficient individually for each player. So that's very similar to the idea of a Nash equilibrium. If, given the strategies of other players, no individual can deviate and find a Pareto dominating bundle, they can't make someone or themselves better without making anyone else worse. We call that a non-cooperative equilibrium, and that will be very important for the goal of this mechanism, because very good properties arise in this equilibrium.

So further, a slightly stronger statement, if it's efficient, this holds for any arbitrary coalition of any size, up to any subset of n players. Then we call it a strong non-cooperative equilibria. So the equilibria is strong if there is no group of players that can form a coalition and create an alternative strategy, that Pareto dominates the original one.

A couple more terminology points here-- we call the non-cooperative equilibria active if each trading post has at least two active buyers and active sellers. So there's trade happening at each trading post, and we call it tight if all active buyers and sellers are quoting the same price.

And feel free to stop me with any questions. There's a lot of notation. Yes.

AUDIENCE: So the strategies can say for one price, I want to buy three goods, and for another price, I want to buy five goods. It's just one price and one quantity.

SAMUEL
BRUCE: Correct. Yes. So now we look at the properties that arise in this non-cooperative equilibria. And I'm going to be using NE throughout for non-cooperative equilibria. It's not Nash equilibria. It's similar, but again slightly stronger. For any market and any negative credit penalty, the active non-cooperative equilibria are all competitive. The tight active non-cooperative equilibria are also competitive.

So all of these game outcomes where coalitions can't do better by deviating, they actually coincide perfectly with the competitive equilibria of the market game. So it's very favorable. And every tight active non-cooperative equilibria is strong, meaning that if it's tight and everyone is quoting the same prices for the active traders, then there are no coalitions that exist that could change a Pareto-dominant strategy.

So there actually exists an application on the blockchain now that implements a lot of Dubey's mechanism pretty faithfully. It's called SPEEDEX. It was developed, published in 2023. It's a decentralized limit order exchange built on a blockchain technology. So it's fully operational, and it exists in a very similar operation. Users submit bids, limit orders to the mechanism. In this case, it's an exchange.

So your bid would look something like I'm willing to trade \$100 for 110 euros, or some currency exchange is what the current application is. And then in each block on the blockchain, the mechanism can calculate the market-clearing prices for each good separately at those trading posts and executes all outstanding orders at that price. So similarly, it's going to group up the limit orders by separate markets for each good. It's going to find the clearing price, and it's going to execute so that all trade that can happen will happen.

So one difference that's very important to note is here, our mechanism is executing all trades at the clearing price-- so the exact intersection price where the buyers and sellers want to sell the same amount. In Dubey's mechanism, remember, all buyers sold at the price they quote. They did not operate at the intersection price. It was at the buyer's price for each limit order. And that may seem unfavorable in Dubey's mechanism, because it gives a lot of power to the sellers, because they extract all of the willingness to pay from the buyers.

When their orders are higher than some intersection price, the seller actually is able to extract the value in between their sell price and the buyer's buy price. However, Dubey proves in his paper that executing at the intersection price breaks down some of the assumptions of the paper, and we can no longer guarantee that the equilibria of the game are competitive. So that is a big problem. Obviously, if the game equilibria is not competitive, it's not a great model for how agents are going to act, and goods are going to clear.

So SPEEDEX operates mostly in the scope of the paper without this big change. And the algorithm behind the scenes is using it's a tatonnement iterative process. That's not extremely important. It's just good to know that it's an iterative strategy. So essentially, at each block, it groups all of the limit orders for each good, and then it's going to iteratively try and quote prices that will clear the market in an algorithm until it finally finds the one that clears the buyers and the sellers.

Some of the outcomes of SPEEDEX-- that it does really well. It has a great runtime per block. Its runtime is of the order of the number of assets squared times the logarithm of the number of outstanding orders. And the logarithm part is very important because you can imagine in most exchange markets or limit order markets, we're expecting a relatively small fixed number of goods. However, there may be many, many, many orders from many, many agents. So an algorithm that scales logarithmically in those number of orders is going to be very favorable for execution.

The clearing prices are also consistent. Remember I said, it's an exchange specifically between currencies. So if you have multiple currencies here-- A, B, C-- there's no arbitrage opportunities between the different currencies. The ratio of prices between A and B times the ratio between B and C is the same as A and C. So there's no opportunity to submit different exchange bids on different goods and extract arbitrage.

And another good thing about having these prices consistent is that all transactions in one block occur at the same prices. Now, that may seem intuitive, but it's actually not how most previous decentralized exchanges are implemented. Most decentralized exchanges will operate on each order as it comes in, ordered in the block, and it opens up to front-running attacks. And what a front-running attack is is essentially, you can imagine some node in the blockchain who has a lot of computing power and therefore validates and proposes blocks very often.

Remember how often a block is-- sorry, how often a node successfully proposes a block is proportional to their computing power. So you may have a specific node who can spy on the transactions that are being submitted in the next block. They see them as they get submitted, and they can add in a transaction of their own, T prime. And if they can reorder the transactions, which they can, because generally, in a block, all transactions can be ordered in any way.

With old decentralized exchanges, they can actually extract extra rent from the transaction they spied on. You can imagine some transaction T wants to buy this currency at this price as a limit order. It's going to execute at this price. But if you submit some transaction that executes just before it, you can actually raise the price they pay a little bit and take a small amount of money in the middle. And this is called a front-running attack, and it's an example of miner-extractable value. So miner-extractable value here just refers to any node who's mining in Bitcoin or other proof-of-work blockchains that has a lot of computing power, has a lot of processing power, can reorder these transactions to benefit themselves and extract extra surplus from other transactions. And SPEEDEX eliminates that opportunity, which is very good.

AUDIENCE: But the Dubey-- is it true that every non-cooperative equilibrium has no arbitrage? There couldn't be one. That has arbitrage.

**SAMUEL
BRUCE:** So in the specific example of Dubey, it's not an exchange market. And again, the prices are competitive, which means that no, there wouldn't be arbitrage opportunities. So looking at SPEEDEX and looking at Dubey's mechanism, we raise some concerns on the actual tractability of finding equilibria.

So right, we've talked about the non-cooperative equilibria, where there's no individual agents or coalitions who can deviate and find Pareto-dominant strategies. And it's very nice in equilibria. However, nowhere in either SPEEDEX or Dubey's proofs do they mention how it can converge on our non-cooperative equilibria. And in general, finding non-cooperative or Nash equilibria, because they're similar concepts, requires each agent to rationally predict the strategies of the other players and submit their optimal bid vector accordingly.

If you think back to our reward function, you're looking at your deviation compared to the strategies of others. So for you to accurately compute and think about as an agent what your strategy should have to be, you able to estimate the strategies of others. And when we're talking about almost continuously valued prices-- I put almost in parentheses here because if we're talking about computing these, it's always going to be finite valued.

But let's say a very, very large number of possible prices, a very large number of goods, a very large number of quantities, the strategy space for each agent, which is the four vectors all over those things, it gets extremely large. And it's very hard to search over. And it's very hard for a rational agent to reason well about what other players are going to submit and what they ought to submit, making it concerning whether or not we would actually find the non-cooperative equilibria of our game.

And even when we iterate many times over the mechanism, it may be completely intractable to just find ourselves in a non-cooperative equilibria. It may be very hard for agents playing in our mechanism to find themselves to our equilibria on their own. So we're going to formalize that a bit more first on what computing a Nash equilibria actually looks like. And we're going to talk about Nash here because it's very well-studied. The non-cooperative equilibria, because of the coalitions, is at least as hard as Nash. So anything that holds here for Nash will hold for that as well.

Nash equilibria to compute them in algorithms-- it's PPAD-complete. And we'll describe what that means exactly in a minute. But importantly, all known algorithms are exponential algorithms. So there are no polynomial time algorithms that can compute Nash equilibria in the numbers of players and strategies. Competitive equilibria are also in the same complexity class, PPAD-complete for general exchange economies, so we can't get around it by finding the competitive equilibria and working backwards to see if it's non-cooperative. That won't work either.

So again, an agent faced with many goods and continuous prices has a very complex optimization problem that computers take exponential time to solve. So convergence is very dubious, and it's very hard to find a situation in which a rational agent will be able to make this optimization themselves.

AUDIENCE: Do we know? Are there a lot of Nash equilibria that are not non-cooperative? Because every non-cooperative equilibrium is a Nash equilibrium with the extra requirement that when you deviate, there is no one that's worse off. So then are there lots of Nash equilibria that would be non-cooperative? Because I mean, presumably, those could arise to for whatever reason.

**SAMUEL
BRUCE:** Yes, so from the definitions, all Nash equilibria are non-competitive and vice versa-- sorry, yeah, non-cooperative, sorry. They're the same in the instance that the deviation player is a single player. The coalition, which we talk about to get to our strong non-cooperative equilibria, is the added constraint. And that would be a specific subset of Nash equilibria.

AUDIENCE: But even if the deviating coalition is a single player, I mean, in a Nash equilibrium, a single player can deviate, and it could make one other player worse off. So it's not necessarily right. So it doesn't mean-- like it doesn't have to be the case that it is-- what were you calling it? What are we calling it? Efficient or Pareto-improving.

AUDIENCE: The Bayes' theorem is that the selfish individual Nash equilibrium without the extra criteria for the game he specifies will be the Walrasian competitive outcome.

AUDIENCE: I see. OK. So that is a result for this game then.

**SAMUEL
BRUCE:** Yes, it's specific to the mechanism.

AUDIENCE: The corollary is competitive equilibria are in the core so they can't be blocked by any subset. So the spirit at least of the extra condition is satisfied once we get to the Walrasian outcome.

AUDIENCE: OK. Does it have this result because it assumes that they can only submit one price and one quantity?

AUDIENCE: I don't know.

**SAMUEL
BRUCE:** Likely. So just some definitions here. Before we go into the formal definition for PPAD. General complexity theory NP is the class of problems that have polynomial time verifiers. And what that means, essentially, is that given the problem and a potential solution, you can find in polynomial time whether or not the solution is correct. So it may take exponential time to search and find a correct solution. But given a potential solution, you can say yes or no in polynomial time. That's the class NP.

A hard problem in a complexity class denotes any problem that is known to be at least as difficult to solve as any other problem in the complexity class. And then further, a complete problem is a problem in the class where any problem can be mapped to that problem. And what that essentially means is if you have a complete problem in a class, you can take any other potential problem, and in polynomial time, a solution to the complete problem can be turned into a solution to the other problem. So a little bit of nomenclature there.

It's worth noting all complete problems are hard. So for a problem to be a map to any other problem in the class, it must be at least as hard as any other problem in the class, and there must be some algorithm that allows it to be converted quickly from a solution of 1 to a solution of another.

And a lot of times in complexity theory, we're looking for these complete problems. Because if you can prove a problem is complete for a complexity class, you know it cannot be simplified any more, and it cannot possibly be more complicated. So it's a very tight bound on exactly how difficult the problem is to solve, because you know it exists as a full member of that class, and we're going to get to it in a second. But Nash equilibria are proven to be complete for this class, PPAD. And that's what we're going to talk about here.

Long, long time ago, Nash proved that every game has a Nash equilibrium, whether in pure or mixed strategies. There is guaranteed to be a Nash equilibria of every game, and simple classes of games, such as two-player, zero-sum games and some graphical games, they have polynomial algorithms to find them. And there's minimax theorems that you can look into. And these are in very specific situations, great, efficient algorithms for solving Nash equilibria. But in the case of general games-- so n players each having their own strategy set and arbitrary utility functions that maps from submitted strategies to utilities-- it's much more complicated.

So they started looking into this a long-- maybe 30 years ago. And for this idea, they created a new complexity class, which is called TFNP or NP total functions. And this is the class of all search problems that are guaranteed to have a solution. So remember, Nash fits in there because we know every Nash equilibrium. Every game has a Nash equilibrium. So we create this complexity class where it's search problems, but there's a guaranteed solution.

However, this is a semantic class, which just means that there are no complete problems for the class. It's too general of a description. Games-- search games with a guaranteed solution is too general to really fit one problem to define the class.

So we break it up into some more specific classes. And one of those classes, PPAD, which stands for polynomial parity for directed graphs-- polynomial parity argument in directed graphs. And it's defined by the complete problem in any directed graph with one unbalanced node.

So you can picture some arbitrary directed graph, and there's one node that's unbalanced, meaning that they're in degree to the node is different than their outdegree. There's guaranteed to be another node in that graph that is also unbalanced. You can imagine if some node has two in and one out that can't come from nowhere. There must also be a node somewhere that has, say, one in and two outs or some other combination of nodes that allows us to get unbalanced. So you can't have an unbalanced node by itself. This is defined as the class PPAD finding that other node given one.

And Nash equilibria was proven to be a member of this class by Daskalakis, Goldberg, and Papadimitriou. Now, I won't take you through the proof of this because it's very long and very complicated. But it's just the intuition here is what matters that we have found like algorithms for Nash, and they are specifically very hard and complete, meaning that no simpler algorithm can exist. And it will always be exponential algorithms to find Nash equilibria under most cases.

So if we know Nash equilibria are extremely hard to compute and we still want to find the solutions to our game and find areas where agents will be able to reasonably find equilibria, we can start looking at alternative types of equilibria because there's more than just Nash. Specifically, we're going to talk about equilibrium concepts that require coordination among the players. Remember, in Nash equilibria, each player is acting on their own. However, if we add a coordinator to the game, it might make things simpler. So instead of each player computing their optimal strategy profile, we're going to add a central coordinator to the game who chooses the strategies of each player for them.

So the coordinator keeps a joint probability distribution across all player strategies-- call that p . And for each iteration of the game, the coordinator is just going to draw from that distribution at random, select some strategy profile for all the players, and is going to tell each player their selected strategy. Each player on their own knows the underlying distribution p , and they know their selected strategy. If I'm the coordinator, I could tell you, like, oh, I'm drawing from this distribution. Here's what I'm telling you to do. I'm not going to tell you what I told the other players to do. So you only learn your own strategy. However, by knowing the underlying distribution, you can then reason about what other players might have been signaled to play.

Under our coordinator mechanism, we will call it a correlated equilibrium when each player, knowing their signaled strategy and the underlying distribution, has no incentive to not play the strategy they were given. So in other words, in our notation here, over all possible strategies of the other players, S_{-i} here, your utility of playing your selected strategy given they're playing that strategy times the probability of that happening must be greater than some fixed deviation S' across again all other possible strategies of each player weighted by the probability of that happening. So each player must individually be incentive compatible to play their strategy. When that's the case, we'll call it a correlated equilibrium.

Now each player here faces the number of strategies squared constraints. And you can see that here because you could get any signal that's in your strategy set. So that's your number of strategies. And for each of those signals, it must be at least as good as any pure deviation. So for each strategy, we have again the number of strategy conditions-- so number of strategies squared for our total constraints on each player. We call it correlated because knowing your strategy gives you information on the possible strategies of other players. So with the coordinator, you're able to correlate players together so that the joint distribution favors playing certain actions together as a group amongst multiple players, even when individually they may not all come to that conclusion.

So this is an example here, and we'll keep bringing this one back up throughout the lecture to demonstrate different equilibria. But we have player one here. Their utility matrix is on the left. And they choose between top and bottom as their two strategies.

Player 2 on the right chooses between left and right, and this is their utility matrix. And then D here just denotes the distribution. So each p is the probability that-- the joint probability of that outcome. So PTL is the probability that player 1 plays t , and player 2 plays L and so on and so forth.

So this is what those constraints would look like for each player in this situation. So player one in this row here, this is the strategy of playing their signal when their signal to play the top row. So weighted average of what is probably going to happen if they play the top row-- so their utility when they play top row and the other player plays left times the probability the other player was signaled to play left plus their probability when the other player plays right times the probability that they were told to play right. And we compare it to any deviation.

So the deviation here for player one, they were told to play top. They could deviate to play bottom. And now the probabilities are the same. Because you were still told to play top.

So it's still the same probabilities that the other player was told to play left versus right. But your utilities would change because now you're playing bottom. So if the coordinator said you play top, they play left, but you play bottom. You get this utility instead. That's why it's here, and then likewise with the right.

We do the same thing for player 1 being told to play the bottom strategy, player 2 being told to play the left strategy and the right strategy. And if all these constraints hold, then the distribution D is a coordinated equilibrium. Moving from correlated equilibria, we're going to talk next about coarse correlated equilibria, which is an even weaker version of equilibria.

So we're in the same setting where we have our coordinator with a distribution drawing strategies for players. However, now the players do not get to learn their selected strategy before committing to the mechanism. So instead of having the information of what you were told to play and using that to guess at what other players might have been told to play, now it's just the case that you must, in expectation over all possible strategies, be better off than some fixed deviation for all possible strategies.

So prior to learning their strategy, each player i must be incentive compatible. And walking through the notation here, this term on the right, this sum, is just the sum. It's the total probability of S negative i because it's summing over all of your strategies the probability of you getting that strategy and then getting that one.

So this is the total probability of the other players playing S negative i . We multiply that times the utility for the deviation, and we sum over all possible other strategies to get the expected utility of any deviation. That's what this right term is. If the expectation of playing the distribution, as you're told, is better or equal to some fixed deviation, we'll call it a coarse correlated equilibria.

And here the number of constraints is actually lesser. So because it's not conditioning on your own strategies, there's only O of number of strategies constraints because it's just your total expectation. Compare it to all possible fixed deviations. So we'll go back to our example here.

We have the same utility matrices and distribution. And our constraints are similar, but they look slightly different. So the top row for each player here is just denoting their expected-- the expected utility that they receive from playing the distribution. So intuitively for player 1, it's three times the probability that this box is chosen plus 1, that one, and so on and so forth. Player 2, it's similar. It's just their utility matrix times the probability. That's where they end up.

And in each case, for each player that expected utility needs to be greater than any deviation to a fixed strategy. So for here that looks like three-- if they deviate to the top row, which is this one here, you have the probability that any left is signaled for player 2 times you play the top fixed plus any deviation-- or sorry, any time the right player is told to play right times your top deviation utility. So again, constraints to make sure that you want to play as told. In this case, it's just in expectation. For each player, we have these constraints, and if they all hold, then it's a coarse correlated equilibria.

Some important things about correlated and coarse correlated equilibria. First, all Nash equilibria are correlated. They're just a subset of correlated equilibria. And likewise, all correlated are a subset of the coarse correlated. And you can think of that in terms of the constraints. They get weaker and weaker. So anything that fits the constraints of a Nash equilibria is going to automatically fit the constraints of a correlated and similarly for coarse correlated equilibria.

Nash equilibrium is actually a special case of correlated equilibria where the probabilities for each player are independent. So you could imagine the coordinator who has his underlying probability distribution, just builds it in such a way that the probability of any joint distribution is just the product of the probability for each individual player playing that strategy. If there's no correlation between the player strategies, then it's a Nash equilibrium, but still in the same space.

The difference between the correlated and the coarse correlated again can be thought of as a difference in commitment and when you commit. In the coarse correlated equilibria, it requires you to commit to the mechanism before you even learn your strategy. So you must think in expectation the mechanism will be better for you, but you don't know what you're being told to play yet. In the correlated equilibria, you get to learn your own signal and then make your decision. So that's why it's a little stricter, because you must be able to learn your signal and still think, yeah, I want to play this. Because the coordinator gives you that strategy and now you're simply just trying to find incentive-compatible allocations, it greatly reduces the complexity for each player's optimization problem.

Now instead of trying to predict the strategies of others, they know this underlying distribution. They can assume that other players are going to play the strategy that they were told, and they just have to form these linear constraints about when it's beneficial for them to play their strategy and when it's not. So it really is helpful when we're talking about our algorithm design because we're no longer talking about our exponential strategy space, but just a very linear space for each player.

One thing is it does require trust in that coordinator. You could imagine some nefarious coordinator that tells you their distribution is p , but it's really some alternate distribution q . And q is really favorable to other players and really favorable against you. You would not want to play in that mechanism. So you must believe that you're receiving fair draws from the distribution that you're playing.

And our blockchain implementations actually kind of inherently solve this issue for us because smart contract code deployed on the blockchain is visible to all participants. And you can actually see by inspecting the code on the contracts what the distribution is, making sure that they're giving you fair draws. And since every player gets to learn the distribution, it's revealing no more information to verify that the mechanism is true.

Importantly, all those constraints for both the correlated and the cross correlated equilibria, they're linear combinations of the strategy probabilities. When our constraints are all linear in terms of our variables, we can easily find the correct distributions that solve our constraints with linear programming methods. These are polynomial time algorithms that allow us to put in our constraints and get out a solution, a distribution that will work for us.

Because we kind of did the math earlier but if we have n players k strategies, you're going to have O of NK squared constraints for correlated equilibria or NK for coarse correlated equilibria. You have the added constraints that probabilities must sum to 1, that they must be greater than zero. But this is a very reasonable number of constraints for a linear programming problem. This is very easily handled by many libraries that exist and many algorithms.

And additionally, one great aspect about linear programming is at no extra cost, you're allowed to add in an optimization to the problem. So instead of just finding some arbitrary distribution that meets all of our incentive compatibility constraints, we can actually maximize over something in the set of distributions that work.

So a classic one is the Pareto planner's problem. Given weights λ for each player, you can maximize the expected utility for each player in a λ -weighted sum. That's a very typical maximization you'd want to do as some coordinator, to try and maximize the λ -weighted sums of utilities of players. And you can do this in linear programming at no extra cost. It's baked into the algorithm already.

This is an example of what that linear program might look like. So again, here we have our maximization problem now. And we're just assuming that the λ weights are one. So this is just you can see some of the player's utilities for top left times the probability of that and so on and so forth for all the other probabilities. We can maximize the expected utility for each player by maximizing this sum, subject to our constraints here.

So these are just our correlated equilibria constraints. I reorganized them a little bit to make it clear that it's very simple to express them just in terms of our probabilities and greater than zero. So it's easy to add these to some matrix format or some constraint algorithm. And just plug this in to our linear program solver, and we'll get a distribution that's correlated or coarse correlated and maximizes the expected utility for the players-- so very promising.

We've been talking a lot about these correlated and coarse correlated equilibria. Why? This polynomial time algorithm is really good compared to the Nash. And here's a little bit more on why that matters.

I said there are Nash equilibria algorithms that are in that class PPAD that are exponential. But if those algorithms exist, why do we really care about how long they take? You could imagine just running it for a really long time and eventually getting a solution to the Nash problem. And Nash are tighter equilibria, and it'll be better probably for the market if we can find that.

However, on our blockchain implementation, those exponential algorithms are very, very hard to use and pretty much are incompatible with most architectures. And the reason why is we'll take Ethereum, for example, because we've been talking about that in terms of smart contracts. When a new block is committed, each node individually runs all of the transactions on that node. And for smart contracts, that looks like all the smart contract updates, the function calls and all the smart contracts, are run individually on every node, across every node in the blockchain.

And to ensure that this can be done in a reasonable amount of time so you can add blocks fairly frequently, there are very strict limits on the complexity allowed by each contract and by each function call. Ethereum, for example, will not let you program any contracts in which you cannot determine how long it will take at compile time. So you can't pass in a variable that determines the length of a loop, or you can't have "while" loops where there's not a guaranteed ending. Anything in programming that has a non-deterministic ending, you can't do in Ethereum, and anything that's more than even low-factor polynomial time complexity gets very, very, very expensive even for known fixed bounds.

And this is true on most blockchains. It differs between them, but most of them require that code is written so that polynomial execution time is guaranteed regardless of what input is passed to the contract. So we need these polynomial time algorithms, and we need them to be fairly simple in order for our mechanism to work. We have our new equilibrium concept, and we want to see if we can use it to help the players induce equilibria in Dubey's game. That's our end goal here, is to find our correlated or coarse correlated equilibria, use that to signal players their strategies, and help induce the non-cooperative equilibria that we found to be so beneficial earlier.

However, because we are working with continuous prices, or almost continuous, and very large strategy spaces, linear programming constraints tend to grow very, very quickly. And it's actually linear in the number of outcomes, which when we're talking about outcomes, that's the possible draws from the joint probability distribution. Those were the variables that we were finding in our linear programming problem. And if you're talking about the joint strategy space in Dubey's game, where each player has many, many, many, many possible limit orders they could submit, the joint probability distribution grows extremely quickly.

So one example for Dubey's game-- two goods, two players, and very restricted on the quantities and prices. So for each good, they're allowed to submit a limit order for quantity between 0 and 9 and for price between 0 and 9. That's it. There's no fractional prices, just integer prices. They have 10 options for each good-- two players, two goods.

The strategy space for each player in this very limited setting is 100 million possible strategies. So if we look at the joint distribution between two players, there's 10 to the 16th joint strategy outcomes. So our linear programming would be optimizing over 10 to the 16th variables. If you add just a third player, that number jumps to 10 to the 24th. So even with polynomial time algorithms across that, our input is scaling exponentially with players and strategies. So there's more optimizations we need to make in order to make this tractable because we want players to be able to submit nearly continuous prices, varying quantities for goods, and we certainly want there to be more than two players. And 10 to the 24th joint strategy outcomes won't do.

We can move into a different way of computing correlated and coarse correlated equilibrium, where our linear programming that we've been talking about explicitly optimizes against all of those joint probabilities. The variables that we're solving is the joint probability distribution for each player, which is a very, very large space because it multiplies for each player. We can use no-regret learning instead, which is a machine learning-based algorithm. And importantly, we drop the optimization over the joint probability distributions and instead only have to optimize over an individual's strategies. So it doesn't grow with the number of players, and it doesn't grow exponentially as quickly as the linear programming does. So we're going to talk about the no-regret learners, get an understanding of them, and then we'll bring it back to the Bayes' mechanism in a few slides.

So no-regret learning-- it's a class of machine learning algorithms, and it's an online algorithm. And an agent is trying to make actions that maximize reward. Like most machine learning things, it takes an action, gets feedback, updates itself. It's online, which just means that there's no training phase like there is for a lot of machine learning algorithms. Instead, it's receiving real feedback and rewards at every time step and making real decisions.

So for each time step-- we'll denote it T here between 0 and big T -- it has to make a decision without knowing the full sequence of inputs. And it has to update itself, and it has to make a decision and receive rewards based off of that.

So we're going to have our action space again. We'll call the actions a now. Still in the same strategy set. And at each time step, it's going to submit an action probabilistically. So it's going to keep some probability vector over how often it wants to play each strategy. It's going to randomly choose one, submit that, and it's going to get feedback from the mechanism on how good it was.

When we talk about how good it was, we're actually not going to use utility. Just in how these algorithms work, they work really nicely. If you use the inverse of utility, which is going to be cost here, and we're actually going to normalize and scale it just to help our algorithm a little bit.

But in every case, our cost here is just a transformation, a linear transformation, on utility. So it's how far under-- so this is our utility of our action-- it's how far under are we from the best possible utility. So you can imagine. If the best utility is really high and we play an action that is really low utility, this is going to be big over the largest possible utility difference.

So you can imagine. If we submit some action that gives us the minimum utility, our cost is going to be one. If we submit some action that gives us really close to our maximum utility, this difference is going to be small, and our cost is going to be close to 0. So it's just a linear transformation on the utility. You can imagine really high utility actions have really low costs and vice versa.

Now we're talking about these algorithms in general, where there can be any arbitrary mechanism or adversary deciding what these costs are. And our cost vector here is just C of t . And at each time period, they're allowed to decide whatever they want for some arbitrary cost that you incur and will update you. And in a lot of cases, you can design these adversaries to play mean against you. They can spy on your strategies and what they think your probabilities are and try and play things that will make you incur high costs.

We are not super interested in those applications. We are specifically interested in the multiplayer setting where we have many agents all trying to optimize individually, submitting to some central mechanism, and that mechanism determines utility for everybody. So while these algorithms are determined for general adversaries, they also do work very nicely in multiplayer games. So they're applicable to our mechanism here.

AUDIENCE: The random action a , that's a single player's action.

SAMUEL Correct.

BRUCE:

AUDIENCE: It says probabilistically. Can it depend on the actions that other players have chosen so as to get that correlation? Or is it always like-- that probability, what does it depend on? Is it completely independent of everything else?

SAMUEL
BRUCE: Agents don't get to learn the actions of others. In fact, the information they receive is very limited. So in each turn, they just get back the cost of their actions. They don't even get to learn their own utility function, or why the costs are as they are, or what other players were doing. It's as limited as possible, but they keep an internal probability. And we'll see in a couple slides how they update that based off of their costs.

So when we're talking about evaluating our regret learning algorithms, we can't really evaluate them against the gold standard, which would be the perfect algorithm that makes all the right choices at all the right times. And a thought experiment we can do to show why that's infeasible is imagine you have some adversary who's picking your costs for you, and at each time step, they pick of all your actions. They're all going to have the maximum cost. They're all going to have a cost one.

And I'm going to pick one action that has cost zero. So that's the best action. If you choose that one, you have no cost. But any other action you choose has the maximum cost. And every single time step, I'm going to randomly change which action is cost zero.

So there's no pattern. There's nothing except one action is better than all the others. You can imagine the perfect algorithm is going to know that, and they're going to find randomly all of these possible actions that have zero cost at every time step. And over our t time steps, they're going to incur a total cost of 0-- perfect algorithm. Even the best possible algorithm online that you could submit to that, intuitively, it's going to be randomly choosing a strategy is as good as anything else because they're randomly choosing which action has cost zero. So you might as well randomize to try and find it. And over t time steps, you're going to have an expected cost of the probability you get that random guess wrong, which is number of actions minus 1 over all of your actions times the number of time steps.

So you're going to grow your cost linearly with the amount of time, and they're going to have cost zero. It's not really a useful comparison here. And you can think of other examples of adversaries, which in retrospect, you can pick some perfect algorithm that would have had awesome cost. But realistically, with an algorithm that's making decisions as it goes, it's not going to perform well at all against that.

When we're evaluating these algorithms, this is a real concern because we want to figure out what makes a good algorithm. And that clearly isn't it, because it's an impractical standard. So instead, we're going to talk about the notion of regret. And what regret is doing is it's comparing what you did for all of the past time steps to what some other better agent might have done.

But we're going to be very restrictive on what the better agent is allowed to do. They're not allowed to just arbitrarily pick whatever they want because they know what the cost-minimizing action was. We're going to have some policy class G , and G is just a class of decisions where we're going to keep it simple. So this is going to map some simple subset of what another agent could have done. All the possible iterations, we'll call that G . Each individual strategy of the opposing agent we'll call π_i .

So now what we're going to do is we're going to look at different types of G s, different restrictions on what the comparison is allowed to do. And we want algorithms that do well against this restricted agent as the number of time steps increases. So we're going to start with external regret, which is a specific version of no-regret learning where our G , our class of alternative strategies, is just fixed strategies. So our comparison agent is restricted to that.

They can only play for all the past t time steps one fixed action across every time step. And G here is going to maximize over that. So we'll call R_G is the cost of the best possible fixed action. Our quantity R_G here, we define it. It's the minimum of all of your actions in your strategy set of the cost of playing that fixed action. That will be our class here for external regret.

R algorithm, which we'll call h , is trying to minimize the difference between our regret and the regret of G , where the regret again, is just the sum of your cost over the time steps from the beginning until now. So we want an algorithm that does well in this difference terms. I.e. Your regret from the actions you played are not much worse than the regret you would have gotten if you had just played some optimal fixed action in hindsight.

And this is a pretty restricted class of comparison agents. They're only allowed to play one strategy for every time step. However, algorithms which use this comparison class actually tend to have really good properties. So one specific algorithm that's used a lot in practice, it's called randomized weighted majority. It minimizes this quantity of external regret. And its difference here between R_H and R_G is bounded over t time steps by the square root of $t \log n$.

N here is the number of strategies-- sorry, a little bit of notation change. But that means per iteration, you're actually decreasing your total bounded regret as t goes on. So as time continues, the amount of regret that you have over the best fixed action on the highest end approaches our fixed regret, and it scales fairly well.

Now this is what the algorithm actually looks like. I'm going to walk us through it. But the important thing I want to stress here is that it's actually fairly simple. And that's one of the best parts of these algorithms is they're fairly simple, which means they run quickly and they run efficiently. Here we have w is the weight for player assigning to strategy i for time step T . L is going to be our cost vector. So L of t minus 1 is our full cost vector over all actions. i denotes the i th good.

It's very important to know for these algorithms, even though the player submits one strategy to the adversary, who gets to decide what the payout is, the adversary doesn't just give them back the cost of the action they did. The adversary actually returns the cost of all possible actions. So you can imagine the cost vector you receive is not individually the cost you received for your action, but a vector of the length of your strategies, telling you what the cost was for all of your possible strategies at that time period. That's going to be important for how the algorithm learns.

Here we're going to fix our losses, our costs to be either 0 or 1. The algorithm works exactly the same for arbitrary costs. It's just the notation gets a little more cumbersome. So we're just going to do 0, 1 loss. And we'll update each weight in each time step by some learning rate η that will denote when we start our training. And then at each time step, given our weights, you will play each strategy i with probability that weight over the sum of all weights.

And you initialize all of your weights equal and all of your probabilities equal so that you randomly select strategies at the beginning. So the player is just going to randomly pick a strategy, submit it to the adversary. The adversary is going to give them a cost vector back of what the cost would have been for any of your strategies, and it's going to take that vector.

And for each good in the vector, if our cost is 1-- so we have high cost-- we're going to decrease our weight slightly. We're going to decrease it by $1 - \eta$. So you can imagine if η was 0.1, we're going to multiply our weight by 0.9. So it's a little less likely we'll play it again. If the cost is zero, it was a good action. We're going to keep our weight the same.

So you do this for all strategies at every time step. And slowly, the strategies that consistently have high cost, their weights are going to decrease. And the probability you're going to play bad actions is going to decrease. And in practice, this actually happens very quickly. And even after just depending on your learning rate, but hundreds to thousands of iterations, you can get it to converge on good strategies for kind adversaries.

So back to economics a little bit. We're now going to think about again the multiplayer setting where the adversary isn't some arbitrary person trying to hurt you. But we have a multiplayer game, and you submit your strategy to the mechanism. Other agents using the same no-regret learning algorithm submit their strategies to the mechanism. The mechanism computes the outcome and informs each agent on their costs, and that would be the cost for the action they submitted but also the costs of all the other actions they could have submitted too. So again, we're still getting a lot of information here.

But in this multiplayer setting, if we take all of the strategies of each player and we take the average of them over all t time steps, we converge to a coarse correlated equilibrium. Just reading through our proposition here, after t iterations of no-regret dynamics, every player of a cost-minimization game has regret of at most ϵ for each of its strategies, where ϵ is our time-averaged regret. We had bounds on it earlier, but it's just the regret you faced over all t time steps divided by t .

We take σ here to be the joint probability space at time t , σ_t . It's the outcome distribution, and we have just σ as the time average. So we take the joint probability distribution of all the agents at each time step, and we average them over all time. This σ is a coarse correlated equilibrium approximated by the size of the regret of each player.

So importantly, remember as players continue to play, their average regret, their regret per iteration goes down. So if we run our agents for many, many iterations, our approximation will get better and better to where the joint strategies of the players are very, very closely approximating a coarse correlated equilibrium. And this is really exciting because now we no longer are trying to optimize over the joint probability distribution. We can take this many, many, many, many dimensional space, break it up. So now each player is just trying to individually maximize over their n strategies. And then the outcome, when we play them all against each other, still converges on our equilibrium-- much less complex in terms of algorithmic compute and we still get reasonably close to our equilibrium.

Now, to move from a coarse correlated equilibrium to a correlated equilibria, we're going to do something very similar in nature to how we did it earlier. Remember that coarse correlated was the class where players, prior to learning their strategies, had to commit to the mechanism. So they just needed to be in expectation better than a fixed action.

And indeed, the external regret, remember, we were comparing to fixed actions. However, in the correlated equilibria, you were given your strategy and still had to be better than any deviation. So you can almost think of, oh, I'm given this strategy, but I'd rather swap to that one. And that's going to be a dual with the idea here, which is a slightly larger comparison class called swap regret.

No-swap regret learning is the algorithms. And it's going to say exactly that. It's the class of all possible mappings where at time t , what if every time I had played i played j ? Instead of a fixed deviation like before from whatever you did to any fixed action, now we're going to say, OK, in retrospect, anytime you played i what if you had played j ? So it's any end-to-end mapping for all of your possible strategies. You can map them arbitrarily to having played a different strategy every time you played that one.

And this is going to be a much bigger and more lenient comparison class is going to be much higher performing. You can imagine actually, the external regret comparison class is a subset of this one, just where you map every action to a fixed action. But now we can do an arbitrary mapping, so we can map every action to whichever action we'd like.

And algorithms for swap regret exist. And again, they're very tractable, and they execute in polynomial time. And like the external regret, if we time-average the probabilities for each player, we come on a correlated equilibrium. So if we run an algorithm that performs really well against this slightly harder class of comparisons, then we converge to a slightly better equilibrium.

We can look at algorithms for both external regret and swap regret, use them, time-average strategies, and get correlated and coarse correlated equilibria, which is really exciting. You can estimate these equilibria without computing the entire joint probability distribution, which was prohibitively expensive, as we saw earlier.

And they work well in multiplayer settings. There's been a lot of research and studies done to prove that they converge. They converge fairly quickly, and they estimate equilibria well.

Now, an interesting thing about them is we lose now the ability to have a maximization function, like we did in linear programming. Remember, we could maximize over the sum of the agents' utilities while doing the linear programming at no extra cost. Now that's kind of been taken away from us, but we'll have to do some investigation to see how much that really costs us.

And again, just like there were polynomial time algorithms for simple Nash equilibria, under simple games, these algorithms perform very, very well. For two-player zero-sum games, they're guaranteed to converge on Nash equilibria for each player. And in fact, more generally, in two-player general-sum games, when players begin with equal weights, they're guaranteed to converge on Nash equilibria.

There are lots of situations, just like with our normal algorithms, where if we simplify the game, then we can find even better converging properties for our no-regret learners. A little example here-- we're using our same players that we were using earlier. We have player 1 and player 2. And we have some graphs here.

I ran this for 20,000 iterations, and this map in the top left is their probability of playing each strategy. So the blue line-- you can see hidden in there somewhere-- is the probability that player zero plays top. Sorry, the letters are off, but that's the blue line is for top. The orange line is the probability they play bottom. The green line is the probability that player two plays left and then red for right.

And as you can see, the strategies, the individual probabilities for each player, they never converge. And that's very common in these games. They are not guaranteed to converge under most situations. They, in fact, go through a lot of these phases.

You can see it where it bounces up and down where it's like they play one strategy very dominantly, and then the other player learns to take advantage of that, exploit it. The one player receives high costs from the exploitation, so they completely switch their strategy and exploit the other player and back and forth. And you see these dynamics where it's just going back and forth and back and forth.

But if we take the time-average of those strategies, this mess seems to converge to something. And in fact, with these probabilities for each player, if we take the joint probabilities-- so it's just probability of top left here, just top times left-- we get strategies that again converge and look like this-- somewhere between 0.0 and 0.1, a couple between 0.10.2, and then the highest one is bottom left with pretty high probability. This turns out to be the unique Nash equilibrium of the game.

I ran it for 20,000 iterations, but we're already starting to converge fairly quickly. So after just some thousands of iterations, we get fairly close to the unique Nash equilibrium of the game with just two of these learners, which is very impressive. Remember, we've been talking about algorithms for the no-regret learning, where they learn at each time step the cost of all of their actions. And we'll just talk about that for a second.

This is called the full information setting. And it is among these classes of algorithms the most information they could receive. They get at each time step the expected cost of any single one of their strategies, and they can use that to update their weights very reasonably well. They get a lot of information.

A slightly less information version is the partial information setting, where again, you get to learn the entire cost vector over all your strategies. But now, instead of getting perfect costs for your strategies, there's some noise in there. In this case, you only get to learn the cost of your strategies based off of what other players did, not what the expected cost might be, as in the full information setting.

And then most restrictive, called the multi-armed bandit setting. The agent no longer receives the cost vector in training. They only instead get the cost of their exact action and the other player's actions. So they just get to learn what they did and how it played out. You no longer get to see any of what-ifs that might have happened if you had submitted a better strategy or worse strategy. You only get to learn and update based off of what actually happened.

And why would we use other information models when the full information one performs so well? It really depends on your mechanism design. So algorithms for those limited in information and multi-armed bandit settings exist but they converge slower, which intuitively makes sense. The agent is getting worse or less information. They're not going to be able to converge as quickly.

However, when you're designing the mechanism, it may be extremely costly to give an agent their entire cost vector. It may be prohibitively difficult to figure out what the cost of potential actions would be. Or you can think of some real-world situations where it's impossible to know what would have happened if you had submitted some alternative action. You only really get to learn what ended up executing. So when you're designing these algorithms that determine the cost for the players, you have to really consider what information is available and easy to compute and choose your information setting accordingly. Again, algorithms exist for all three types of information. However, the less information you give the learning agent, the slower it may converge.

Now looking back towards Dubey's mechanism, we're really worried about these large action spaces. That was what drew us to these no-regret learners is, OK, we have a lot of strategies, continuous prices. How can we handle that? How we reconcile that when the strategy space is extremely huge? And no-regret learners are actually capable of doing that.

So algorithms exist for no-regret learners on completely continuous action spaces like prices and very large action spaces like our possible strategy sets, even under limited information settings like our bandit setting where you only learn the cost of the action you actually submitted. So these algorithms exist. They are fairly new and improving very quickly, but they can do in relatively polynomial time these optimizations that converge on correlated and cross-correlated equilibria without having to restrict the strategy space at all, which is very impressive.

Just to summarize regret learning a little bit because I know there's a lot of information, and they're a little complicated. They're online algorithms, which means that they have to take in information and costs and make decisions as it goes. They don't train and then test. It's all they update as they perform. Our external regret learners compare error costs to the best fixed strategy in retrospect and optimize against that. Our swap regret learners minimize cost over the best fixed mapping between strategies, and the time-averaged strategies for external regret and swap regret converge on correlated and coarse correlated equilibria. Now, they can make these estimations without computing the entire joint probability distribution, which is a huge algorithm improvement over our linear programming, which had to optimize the entire distribution.

Kind of bringing it all together-- we've kind of gone in a lot of different directions here, but the general roadmap for the work that we're doing together involves pieces from all of this. So we want to design regret learning agents and a training mechanism that can efficiently estimate these equilibria for Dubey's game. For what strategies? What limit order should be submitted to the mechanism at each time period?

We want to use this equilibrium as a distribution for a coordinator, and we want to be able to deploy this coordinator with the mechanism for implementing Dubey's market onto the blockchain. And the goal is to get algorithms in place for these no-regret learners that can effectively compute what the equilibria should look like and importantly, what it looks like under varying conditions. You could imagine if you train your no-regret learners under a very specific endowment and a very specific endowment of other players, it may find solutions that are very good for that one iteration of the market.

But how to adapt those learners when they face shocks to their endowment? They face shocks to their utility functions. Other players face shocks that change the market. How you adapt these algorithms is a very complicated and interesting point of study, which is what we're focusing on right now.

But we want to build these regret learners. We want to be able to estimate equilibria for Dubey's mechanism. And if we can efficiently deploy them onto the blockchain, then we'll have a functioning example of how a lot of the things we've talked about in this class can be put into practice, used effectively in a real-world example on the blockchain-- so very exciting for us working towards it every day.

I guess I'll end off with just some outstanding questions that we're really actively investigating, just to think about what the next steps past all of this is. So what were some of the things we're very curious about is what is the efficiency of the outcomes computed by these no-regret learners? Remember, we lose the ability to maximize an objective function when we lose the linear programming solution. So it's still ongoing exactly for Dubey's mechanism, how optimal and efficient these learners are. Even if they do find an equilibrium, are there ones that are better for certain agents than others? And then are there ways that we can modify our training algorithm that improve the outcomes we see?

And then again, like we talked about shocks-- how can we change the distribution? How can we have the agents be dynamic to new agents entering the market, shocks to their endowment, shocks to their utilities? When things change, can these agents predict that in the market, instead of just being very static for a very certain configuration?

And then, lastly, what's the optimal information setting for these learners? We talked about the full information, the limited information, and the multi-armed bandits, but there's trade-offs when implementing them. Because when you talk about the ones with more information, they converge faster, but it may take much more time per iteration because you have to calculate the cost over all of their possible strategies, which can be hard or prohibitively expensive. On the other end, you have the multi-armed bandits, which may be very, very quick per iteration because you just need to see what the cost of the realized state of the world was, but it may take far too long for them to find appropriate strategies. So where the trade-off is in there, how we can make that work-- that's also a question we hope to answer.

I think that's just about it. I hope that was very informative, and that we were able to track it. I know we talked about a ton of different things but all fitting together for this specific limit-order market. And hopefully, we can get it implemented. Thanks.