Developing an Automated Market Maker on the Algorand Blockchain

Haowei Fu*, Jack Gardner*, Jingwei Huang†, Pratik Pugalia*,
*School of Computer Science
Carnegie Mellon University, Pittsburgh, USA, {haoweif, jhgardne, ppugalia}@andrew.cmu.edu
†Tepper School of Business
Carnegie Mellon University, Pittsburgh, USA, jingwei6@andrew.cmu.edu

***Abstract*—Automated Market Makers (AMMs) are a popular application in Decentralized Finance (DeFi) that allow individuals to exchange cryptocurrencies by interacting with smart contracts. In this work, we describe the core functionality of AMMs and outline a series of modifications we made to an example AMM created by Algorand's Developer Relations group. Developing in PyTeal and Beaker, we modify the existing AMM that uses a Constant Product Market Maker (CPMM) model to support a Concentrated Liquidity Market Maker (CLMM) model, allowing Liquidity Providers (LPs) to provide liquidity within defined price ranges, called ticks. With our modifications, we showcase the core AMM operations of minting, burning, and swapping, but further development work is necessary to let users define granular ticks and to ensure the change in price is accurately reflected in the amount of liquidity provided within each tick.**

***Index Terms*—Automated Market Maker, Concentrated Liquidity, DeFi, Blockchain**

I. Introduction

Decentralized Finance (DeFi) is a growing industry that utilizes emerging smart contract technology to create financial products and services without the need for a centralized trusted party [10].

Automated Market Makers (AMMs) are a type of DeFi application that support decentralized exchange of cryptocurrencies [17]. AMMs support decentralized exchange through the creation of liquidity pools where individuals can provide capital to facilitate trades instead of a traditional order-book exchange that relies on buyers and sellers. Smart contract technology makes possible the creation of liquidity pools and allows the developers of AMMs to set prices for the assets that run on these decentralized exchanges using self-executable algorithms. Anyone can supply liquidity into a liquidity pool by depositing both assets represented within the pool, and as incentive, AMM applications reward liquidity providers (LPs) with a portion of fees paid for transactions executed in a pool. While AMMs automate the trading process, liquidity providers are necessary for AMMs to function properly. As more liquidity is provided, AMMs become less prone to price slippage that can occur when trades are executed.

A new AMM implementation, called a concentrated liquidity market maker, allows LPs to add liquidity within a specified price range, e.g., a price range where assets are often traded, to facilitate more efficient use the capital they provide. Uniswap, a popular AMM on the Ethereum blockchain, implements a concentrated liquidity model in their version three protocol [2]. In Uniswap, price ranges are defined by "ticks" that LPs provide as an input when they provide liquidity to the platform. With the ability for LPs to provide liquidity in concentrated price ranges, the likelihood that price slippage would occur in common price ranges for an asset pair decreases greatly, and LPs can earn more fees for a relatively smaller amount of capital contributed compared to standard AMM applications.

In our work, we closely follow the design of Uniswap v3 [2] to work towards a concentrated liquidity AMM for the Algorand Blockchain [3]. Algorand's most-used AMM application, called Tinyman [15], uses the constant product market maker model and is written in TEAL [14], the assembly-style language native to Algorand. The modifications towards a concentrated liquidity AMM that we implement build upon an existing constant product market maker that is written in PyTeal [13], a Python wrapper around TEAL, and Beaker [7], a PyTeal framework that supports easier deployment and interaction with smart contracts on Algorand. Our goal in modifying this AMM is to provide the features of a concentrated liquidity model on Algorand and to better promote contribution from the Algorand community by developing with PyTeal and Beaker.

II. Background on Decentralized Finance and Automated Market Makers

With the development of blockchain networks and associated smart contract technology that can handle increasingly more complex computational and storage requirements, Decentralized Finance (DeFi) has emerged as an industry that offers financial products on the blockchain. Blockchain technology supports the settlement of transactions and smart contract technology allows users to interact with financial services and execute transactions [10]. Decentralized finance has gained popularity due to the ability for service providers to issue immutable smart contracts that are available to the public for increased auditability and transparency. Further, DeFi applications are open to anyone with a crypto wallet and allow users to maintain custody over their assets as they can interact with decentralized apps (dapps) directly [8].

A. Decentralized Exchange through Automated Market Makers

Lending, exchanges, payments, and asset management are four primary financial services that are now available as dapps. Automated Market Makers (AMMs) are used to support decentralized exchange by linking buyers and sellers to funds stored in smart contracts [10].

AMMs and their associated smart contract technology are often described as Constant Function Market Makers, including popular applications such as Uniswap, Balancer, and Curve [6]. The most common type of constant function AMM is the constant product market maker, which uses the invariant formula of $x * y = k$, where x represents the quantity of one asset, y represents that of another asset, and k is a constant value that represents the total liquidity available in an asset pool. The primary disadvantage of Constant Product AMMs is that they may be prone to price slippage in cases where there is limited amount of capital. Constant product market makers also only have one liquidity pool per asset pair, and if the value of a token in the pool increases (decreases) enough, the liquidity provided by a Liquidity Provider (LP) may be insufficient to facilitate a trade, rendering their capital idle. However, sudden changes in asset prices create an arbitrage opportunity for LPs, which leads assets traded on AMMs to revert to their market prices [10]. Uniswap's v3 [2] improves capital efficiency and reduces the chance of price slippage by allowing LPs to provide liquidity in concentrated price ranges.

All AMM applications require LPs to deposit funds into a liquidity pool comprised of two or more assets that allow users to trade assets in the pool, and the algorithm defined in each application's smart contract calculates asset prices based on the quantity of assets available in the liquidity pool. In exchange for providing liquidity, many AMM solutions offer LPs liquidity pool tokens that act as a receipt for the assets deposited and provide a mechanism for them to accrue interest on their deposit through fees that AMMs applications charge for transactions [18].

B. Algorand

The Algorand blockchain was deployed in 2019 and runs on a proof of stake consensus network [4]. Algorand supports applications powered by smart contracts and has attracted DeFi applications with a rich community of users [9], but the platform's current AMM solution, Tinyman, has not received significant adoption compared to AMM solutions on other blockchain networks (see table I), and we believe Algorand could benefit from an improved AMM solution that uses a concentrated liquidity protocol.

III. Automated Market Maker Core Functionality

In this section, we provide a general description of the core operations that Automated Marker Maker (AMM) applications support: minting liquidity, swapping assets, and burning liquidity. While specific AMM applications vary in their implementation, these operations are common to all AMMs and make decentralized exchange possible.

Name	Chain	Total Value Locked (USD $B)
Curve	Multichain	3.61
Uniswap v3	Multichain	2.52
Balancer	Multichain	1.48
Tinyman	Algorand	0.00935

TABLE I
AMM Applications by Total Value Locked.
Source: DeFiLlama [11]

A. Minting Liquidity

AMMs require Liquidity Providers (LPs) to issue capital into liquidity pools, usually comprised of two assets, to facilitate trades. Liquidity providers must issue capital for the assets in a liquidity pool based on the current ratio of the liquidity in a pool so that the additional liquidity added doesn't change the price in the pool. By contributing assets to a liquidity pool, LPs gain liquidity tokens as a reward that represent their contribution to the liquidity pool and allow them to earn transaction fees, the main incentive for a LP to contribute liquidity. This process of providing liquidity is called minting in AMM applications.

B. Swapping Liquidity

Once LPs have provided liquidity into a liquidity pool, a user interacting with an AMM can swap one asset for another at an exchange rate determined by the ratio of the two assets available in the pool and the amount of assets the user intends to swap [12]. The exchange rate must incorporate the amount of assets the user wants to swap so that the user pays the updated exchange rate based on their swap.

When users swap, there is the potential for price slippage to occur if the amount of liquidity provided for assets in a pool is low or if frequent swapping in preference of one asset over another asset occurs over a short time frame. For instance, if swappers prefer asset A over asset B, the ratio of A:B will keep increasing as swappers exchange asset B for asset A. When the asset ratio, i.e., the exchange rate, change rapidly, price slippage happens.

C. Burning Liquidity

Just as LPs can provide liquidity, they also have options to withdraw the liquidity they have provided by exchanging the liquidity pool tokens they received when minting liquidity. This process is called Burning. When LPs want to remove their contribution from the pool, they return their liquidity tokens to the liquidity pool and receive assets from the pool based on the pool's current exchange rate. To receive a certain ratio of assets, LPs may choose to swap assets in the pool to reach a desired exchange rate before returning their liquidity tokens.

IV. Concentrated Liquidity

To address inefficiencies in the use of capital from Liquidity Providers (LPs), a concentrated liquidity market maker (CLMM) allows LPs to provide liquidity in defined ranges which are then aggregated into a single liquidity pool and allow users to interact with a CLMM in the same way they would with a Constant Product Market Maker (CPMM). By allowing LPs to provide liquidity in specified ranges, more liquidity can be maintained in common price ranges for asset pairs, leading to both improvements in the use of LPs capital and decreased chance of price slippage for users interacting with a CLMM. In this section we discuss how CLMMs implement concentrated liquidity, our work towards a CLMM on Algorand, and the benefits for end-users than interact with CLMMs.

A. Features of Concentrated Liquidity

Using the terminology of Uniswap's V3 protocol [2], asset price ranges are demarcated by discrete "ticks" which provide endpoints for the price range in which LPs can contribute their capital. In Uniswap, ticks are defined at every price p that is an integer power of 1.0001, which means that each tick represents a 0.1% (1 basis point) price movement and leads to the following definition of price at a tick index i:

$$p(i) = 1.0001^i \tag{1}$$

Each price range has a specified amount of capital, and within a price range, the liquidity pool functions in exactly the same way as in the CPMM model. The asset price can deviate from the range in which a LP has provided capital when the capital the LP provided is swapped for another asset, and the LP's capital is now composed of a single asset whose value decreased.

B. Example Liquidity Provision and Swap Scenario

Consider a scenario, pulled from the Uniswap V3 development book [16], where a LP wants to provide liquidity to a liquidity pool with an asset pair of two coins, coin X and coin Y, and make a trade, "swap", between the two coins only within a defined price range. The price of coin Y is \$5000 per 1 coin X, and the LP wants to purchase coin X. Since buying coin X will drive the price of coin X higher in the liquidity pool, the LP needs to provide liquidity in a price range that includes the price of coin X after this swap. This means that the liquidity the LP provides must be based on the maximum change in the quantity of coin X and coin Y. Following the example, let the lower price be \$4545, and let the upper price be \$5500.

Uniswap's implementation tracks the square root of price to avoid rounding errors when tracking the liquidity available within asset pools, but we will use the nominal price value here for simplicity.

Knowing the lower price, current price, and upper price of the range in which the LP wants to execute a swap, we can calculate the amount of liquidity the LP must provide based on the maximum price change within the range:

$$L = \Delta x \frac{P_l P_c}{P_l - P_c}$$

and

$$L = \frac{\Delta y}{P_c - P_u}$$

Plugging in our prices, we get:

$$L = \Delta x \frac{P_l P_c}{P_l - P_c} = 1 coinX * \frac{4545 * 5000}{5000 - 4545} = 413,182, \text{and}$$

$$L = \frac{\Delta y}{P_c - P_u} = \frac{5000 coinY}{5500 - 5000} = 10$$

Since we want to purchase coin X, we'll use the lower liquidity amount to calculate the change in price for providing coin Y in exchange for coin X:

$$\Delta P = \frac{coinY}{1449.0}. \tag{2}$$

Swapping 50 coin y, we get

$$\Delta P = \frac{50Y}{10} = 5.$$

Adding this to the current price, we get

$$5 + 5000 = 5005$$

Calculating the price change of coin X, we have:

$$\Delta \frac{1}{P} = \frac{1}{5500} - \frac{1}{5000} = -0.000018.$$

We can now calculate the change in quantity of coin X:

$$\Delta X = -0.000018 * 10 = -0.00018 coinX$$

, which is negative as we're removing it from the pool.

We provided a simple example of a swap above, but when an asset's price crosses a tick, the available liquidity updates based on the sum of the reserves provided by LPs in the new price range of the asset. After a tick is crossed, the swap continues just like the above example until it reaches the next price range where a LP has provided capital.

V. Contributing to an Example AMM

In this section, we discuss our implementation of a concentrated liquidity market maker on Algorand.[1] Our implementation is incomplete and has a number of limitations, which are discussed in detail in section VI. To work towards a concentrated liquidity market maker, we build upon an existing implementation of a constant product market maker created by Algorand's developer relations group. [2] This constant product market maker supports the core operations of minting, burning, and swapping for any pair of Algorand Standard Assets.

We now walk through our development work by discussing the global states available, and the modifications we make to the three core functions.

Global states: We implemented a `ReservedApplicationStateValue` that keeps track of the amount of assets of A and B within each tick range itself. This was coupled with a tick index, which keeps track of the current tick range where liquidity is active.

1) Minting:

[1]https://github.com/AlgorandAMM/clmmamm
[2]https://github.com/algorand-devrel/beaker/tree/master/examples/amm

Instead of minting to provide liquidity to the entire pool as in a Constant Product Market Maker, LPs now now choose a specific tick that represents a price range in which they want to provide liquidity, supporting swaps only within that price range. When LPs call the mint function, this means they specify the tick index that allows them to mint liquidity within a specified price range.

2) Burning:
The logic behind burning is similar to that of the existing AMM implementation. LPs that call burn receive the liquidity that they initially provided from the pool that they put tokens in, plus transaction fees earned from swaps executed in the pool.
3) Swapping:
Swapping was the core of the logic we implemented. Firstly, swaps would begin from the pool at the current tick. The amount of asset that needed to be swapped would be calculated, and if it did not surpass what was available at the current pool, swapping executes in that pool using the constant product formula. However, if the amount surpasses the available liquidity in the current pool, then we would proceed to continue the swap using the liquidity in the next tick with available liquidity, continuing until the amount needed to be swapped was removed. If, there isn't enough liquidity in the whole contract to support the transaction, after we reach the last pool, the contract would hit an error and revert completely (due to the assertion clause after the code was translated to TEAL).

We then proceed to demonstrate the functionality in our `main.py` file. We run this demo by deploying the modified AMM in Algorand's Sandbox [5], a testnet that we can run locally on our computers.

The following are the operations we execute to fully test the logic behind our application:

1) Bootstrap the pool (Generate the initial asset pair, deploy the contract in the Sandbox testnet, and get the contract address).
2) Fund the pool initially, such that our pool and participant have enough funds to execute swaps.
3) Mint 4 times in different ticks, representing 4 different price ranges. Note that we execute 4 mints for the purpose of demonstration, but minting is possible within any number of defined ticks. 16 ticks can be defined using Algorand's standard `ReservedApplication-StateValue`.
4) Swap within a tick and across ticks both ways from Asset A to B and B to A.
5) Burn to return liquidity provided in a tick to a LP.

VI. Limitations

With the modifications we've made, each tick supports swapping a pre-defined amount of assets, and due to the finite number of ticks and their corresponding swap amounts that we define in our concentrated liquidity model, we can only support trades of limited asset amounts. Additionally, we do not impose requirements on the liquidity that is provided within a tick. As a result, the ratio of assets available across ticks do not accurately reflect a change in price from one tick to another. To do this properly, each tick should maintain a ratio of assets that corresponds to the change in price from the previous tick to the current. With the modifications we have made, each tick supports swapping a pre-defined amount of assets, and due to the finite number of ticks and their corresponding price ranges that we define in our concentrated liquidity model, we can only implement swaps within these price ranges. Additionally, we do not impose requirements on the liquidity that is provided within a tick. As a result, the ratio of assets available across ticks do not accurately reflect a change in price from one tick to another. To do this properly, each tick should maintain a ratio of assets that corresponds to the change in price from the previous tick to the current.

As mentioned in the previous section, the `ReservedAp-plicationStateValue` can store only 16 values, meaning that our current implementation allows us to only define 16 different ticks. To extend the number of ticks, we would modify our implementation to support Box Storage [1], a new feature for Algorand smart contracts that allow contracts to store up to 32KB worth of data.

VII. Benefits of Concentrated Liquidity

Concentrated liquidity brings with it a number of key benefits for Liquidity Providers (LPs) and swappers trading in its liquidity pools.

For LPs, the capital they provide to a liquidity pool can be used more efficiently than they could be in a constant product market maker. In constant product markets, the liquidity an LP provides is distributed evenly across the entire price range for a given asset pair. In concentrated liquidity models,however, LPs can contribute liquidity within a specified price range. This flexibility allows LPs to choose price ranges that are common for a given asset pair. This new feature provides greater liquidity within those price ranges and require less capital from LPs to do so. This is especially relevant for LPs that provide liquidity for asset pairs that trade in narrow price ranges.

When liquidity is provided into a specified price range, the capital is fully effective within that price range and allows LPs to earn fees on any transaction carried out within that price range. In the constant product model, since liquidity is even spread among all possible price ranges, only a fraction of the provided liquidity is available at a given price range.

For swappers, the greater concentration of liquidity within a price ranges decreases the chance of price slippage when executing a trade. While swaps within a price range are executed using the constant product formula, the greater concentration of liquidity within a price range means that the change in liquidity causes a relatively lower change in the ratio of the quantity of the asset pair being traded and as a result leads to lesser price change.

VIII. Challenges

Although PyTeal and Beaker allow developers on Algorand to write smart contracts using functional programming abstractions over the low-level language TEAL, developing in PyTeal and Beaker remains a difficult process. In particular, debugging code written in PyTeal remains challenging as the error messages provided are for the compiled PyTeal code in TEAL.

Another key challenge is the need to accurately track and respond to changes in the price of the underlying assets being traded. This involves monitoring the the trading activity in an asset pair's liquidity pool to understand changes in ticks, which are the smallest possible change in price, and using this information to maintain the market maker's internal pricing mechanisms. This can be a complex process, as it requires the Automated Market Maker (AMM) to handle large amounts of data in real-time and make rapid calculations to determine the appropriate price for each trade. Additionally, the AMM must be able to handle the volatility of the market and adjust its internal pricing mechanisms accordingly to ensure that it remains competitive and fair to all market participants. Overall, managing the complexities of ticks and maintaining accurate price changes is a critical aspect of developing a successful concentrated liquidity AMM.

IX. Conclusion

The modifications we outline in this paper are a step towards a concentrated liquidity market maker on the Algorand Blockchain. Utilizing new features such as Algorand's Box Storage and ensuring that our modified contract has the ability to track changes in liquidity reserves and price ranges are the two key next steps that would bring the current implementation closer to a minimum working example. While our implementation is in progress, we outline the challenges we encountered in building on Algorand and offer direction for developing this AMM further. We also provide concrete examples to introduce the core functionality of AMMs to help readers better understand this important DeFi application.

Acknowledgment

We thank Professor Nicolas Christin for advising us throughout this project and for helping us scope our modifications to Algorand's example Automated Market Maker. We also thank Arjun Brar for sharing his expertise in PyTeal and Beaker.

References

[1] *(BETA) Smart Contract Storage: Boxes.* https://developer.algorand.org/articles/smart-contract-storage-boxes/.

[2] Hayden Adams et al. "Uniswap v3 core". In: *Tech. rep., Uniswap, Tech. Rep.* (2021).

[3] *Algorand.* https://www.algorand.com/.

[4] *Algorand Core Protocol.* https://www.algorand.com/technology/algorand-protocol.

[5] *Algorand Node Sandbox.* https://github.com/algorand/sandbox.

[6] Guillermo Angeris and Tarun Chitra. "Improved Price Oracles: Constant Function Market Makers". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies.* AFT '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 80–91. ISBN: 9781450381390. DOI: 10.1145/3419614.3423251. URL: https://doi.org/10.1145/3419614.3423251.

[7] *Beaker.* https://algorand-devrel.github.io/beaker/html/index.html.

[8] *Blockchain for Decentralized Finance DeFi.* https://consensys.net/blockchain-use-cases/decentralized-finance/.

[9] *Building an interoperable ecosystem.* https://ecosystem.algorand.com/.

[10] Francesca Carapella et al. "Decentralized Finance (DeFi): Transformative Potential & Associated Risks". In: *Finance and Economics Discussion Series.* Washington: Board of Governors of the Federal Reserve System, 2022.

[11] *DeFiLlama.* https://defillama.com/.

[12] *Design Doc.* https://docs.tinyman.org/design-doc.

[13] *PyTeal.* https://pyteal.readthedocs.io/en/stable/index.html.

[14] *The smart contract language.* https://developer.algorand.org/docs/get-details/dapps/avm/teal/.

[15] *Tinyman.* https://tinyman.org/.

[16] *Uniswap Development Book.* https://uniswapv3book.com/.

[17] *What Are Automated Market Makers.* https://www.gemini.com/cryptopedia/amm-what-are-automated-market-makers.

[18] *What Are Liquidity Pool (LP) Tokens?* https://academy.binance.com/en/articles/what-are-liquidity-pool-lp-tokens.