Distributed Algorithms for the Housing Allocation Problem

1st Tong-Nong Lin
Cockrell School of Engineering
University of Texas at Austin
Austin, TX

2nd Eric Wang
Cockrell School of Engineering
University of Texas at Austin
Austin, TX

***Abstract*—This paper presents an implementation of the Top Trading Cycle (TTC) algorithm, a method for achieving Pareto optimal and core-stable allocations in the housing allocation problem, where each agent has strict preferences over indivisible resources. Using socket programming in C++, we simulate a distributed environment where each agent communicates asynchronously to exchange preferences and execute trades. For efficient cycle detection, we employ a modified, deterministic version of the Las Vegas algorithm based on a cycle-finding technique for functional graphs. This approach preserves the TTC algorithm's guarantees while optimizing communication and computation. We evaluate the performance of our implementation and demonstrate the effectiveness of deterministic cycle detection in distributed resource allocation.**

***Index Terms*—Top Trading Cycle (TTC), Housing Allocation Problem, Distributed Systems Simulation, Pareto Optimality, Core Stability.**

I. Introduction

In resource allocation problems, ensuring fair and efficient distribution of indivisible goods among agents with specific preferences is a fundamental challenge in economics and computer science. The *housing allocation problem*, introduced by Shapley and Scarf [1], is a classic example where each agent initially owns a single indivisible good (house) and has strict preferences over all houses. This problem models situations where direct exchanges among agents can lead to mutually beneficial outcomes without monetary transactions. Applications extend to various real-world scenarios, including organ exchange programs [2], school choice mechanisms [3], assignment of offices or tasks within organizations, and allocation of shared computational resources in distributed systems.

Solving the housing allocation problem efficiently and fairly is crucial because it directly impacts the satisfaction and utility of the agents involved. For instance, in organ exchange programs, an efficient allocation can save lives by optimally matching donors and recipients. In school choice programs, fair allocation ensures that students are assigned to schools in a manner that respects their preferences and promotes equity. In distributed computing, allocating resources like processing power or storage optimally can significantly enhance system performance and user satisfaction.

Solutions to this problem must satisfy several core criteria:

- **Pareto Optimality**: No other allocation can make at least one agent better off without making another agent worse off.
- **Core Stability**: No subset of agents can reassign goods among themselves in a way that makes all of them strictly better off.

The *Top Trading Cycle* (TTC) algorithm, introduced by Shapley and Scarf [1], addresses these criteria by repeatedly forming and executing cycles of trades based on agents' top preferences. This mechanism ensures that each agent ends up with the most preferred good available to them while guaranteeing a Pareto optimal and core-stable allocation. Due to its desirable properties, the TTC algorithm has been widely studied and applied in various centralized settings.

However, implementing the TTC algorithm in a distributed environment poses significant challenges, particularly in managing asynchronous communication and reliable cycle detection among autonomous agents. In distributed systems where agents operate independently and lack global knowledge, coordination becomes complex. Addressing these challenges is essential for applications in decentralized multi-agent systems, such as peer-to-peer networks, distributed ledgers, and collaborative resource-sharing platforms.

A. Problem Statement

Traditional implementations of the TTC algorithm assume a centralized coordinator with complete information about agents' preferences and current allocations. This centralization may not be feasible or efficient in large-scale systems where agents are geographically dispersed or operate under privacy constraints. Implementing TTC in a distributed environment requires agents to communicate and coordinate without centralized control, ensuring that the algorithm's theoretical guarantees are preserved.

Key challenges in a distributed implementation include:

- **Asynchronous Communication**: Agents communicate over networks with latency and possible message loss, requiring robust protocols to ensure consistency.
- **Reliable Cycle Detection**: Identifying trading cycles without centralized oversight necessitates distributed algorithms that can detect cycles deterministically.

- **Scalability and Efficiency**: Minimizing communication overhead and computational complexity is critical for the algorithm to be practical in large systems.

In this work, we propose a distributed implementation of the TTC algorithm using socket programming in C++. Agents are modeled as autonomous nodes that asynchronously exchange preference information and execute trades through real-time communication channels. We address the challenges of distributed cycle detection by employing deterministic algorithms adapted from cycle detection methods in functional graphs.

B. Related Work

The TTC algorithm has been extensively studied in the context of market and mechanism design [4], [5]. Roth and Postlewaite [6] analyzed the strategic properties of TTC, demonstrating its strategy-proofness in certain settings. Abdulkadiroğlu and Sönmez [3] applied TTC to school choice programs, leading to practical implementations in cities like Boston and New York.

Distributed implementations of resource allocation mechanisms have gained attention with the rise of distributed systems and multi-agent networks. Previous research on distributed TTC implementations is limited. Some works have explored probabilistic methods for distributed cycle detection, such as using random walks or Las Vegas algorithms [7]. However, these approaches may yield inconsistent results and lack determinism, which is undesirable in critical applications.

Fischer and Michael [8] studied distributed algorithms for matching problems but did not focus specifically on TTC. Nguyen et al. [9] proposed distributed mechanisms for resource allocation in cloud computing but relied on approximation algorithms. Our work distinguishes itself by providing a deterministic distributed implementation of the TTC algorithm suitable for asynchronous environments. We adapt deterministic cycle detection algorithms from graph theory [10], ensuring reliable and consistent outcomes. To our knowledge, this is one of the first implementations that combines the TTC algorithm with deterministic distributed cycle detection in an asynchronous setting.

C. Importance and Impact

Addressing the distributed housing allocation problem is important for several reasons:

- **Decentralization**: In many modern applications, centralized control is either impractical or undesirable due to scalability issues, single points of failure, or privacy concerns.
- **Autonomy**: Agents may represent independent entities with their own preferences and decision-making capabilities, necessitating algorithms that function without central coordination.
- **Real-World Applications**: Decentralized marketplaces, peer-to-peer networks, and collaborative platforms benefit from efficient resource allocation mechanisms that respect individual preferences.

For example, in decentralized energy markets, households with renewable energy sources can trade surplus energy. Efficient and fair allocation algorithms ensure that energy is distributed where it is most needed while respecting participants' preferences [11]. In distributed computing, tasks and computational resources need to be allocated among numerous nodes; efficient algorithms improve overall system performance [12].

D. Contributions

The main contributions of this paper are:

1) **Distributed TTC Algorithm**: We present a distributed implementation of the TTC algorithm using socket programming in C++. Our implementation allows agents to operate asynchronously, communicating only with relevant parties to share preferences and execute trades.
2) **Deterministic Cycle Detection**: We adapt deterministic algorithms for cycle detection in functional graphs to the context of distributed systems. This ensures that cycle detection is reliable and consistent, even in asynchronous environments with network delays.
3) **Scalability and Efficiency**: Our approach optimizes communication and computational overhead, making it suitable for large-scale distributed systems. We analyze the performance of our implementation through simulations.
4) **Real-World Applicability**: We discuss the relevance of our work to real-world applications, such as decentralized marketplaces, resource sharing in peer-to-peer networks, and distributed ledgers.

E. Organization of the Paper

The remainder of this paper is organized as follows: Section II provides a detailed description of the TTC algorithm and the modified deterministic cycle detection method. Section III details the design and implementation using socket programming in C++. Section IV presents experimental results and performance evaluations. Section V discusses the implications of our work and potential future directions.

II. The Top Trading Cycle Algorithm and Cycle Detection Methods

The Top Trading Cycle (TTC) algorithm, introduced by Shapley and Scarf [1], provides a robust solution to the housing allocation problem, ensuring both Pareto optimality and core stability. This section delves into the TTC algorithm, offers a detailed illustrative example, and discusses essential cycle detection methods for implementing TTC in a distributed environment.

A. The Top Trading Cycle Algorithm

1) Problem Formulation: Consider a set of n agents $A = \{a_1, a_2, \ldots, a_n\}$ and a set of n indivisible goods (houses) $H = \{h_1, h_2, \ldots, h_n\}$. Each agent a_i initially owns exactly one house h_i and has a strict preference ordering $\succ_i$ over all houses in H. The objective is to find an allocation $\mu : A \to H$

that reassigns houses to agents based on their preferences while satisfying:

- **Pareto Optimality**: There is no other allocation μ' such that for all $a_i \in A$, $\mu'(a_i) \succeq_i \mu(a_i)$, and for at least one a_j, $\mu'(a_j) \succ_j \mu(a_j)$.
- **Core Stability**: There is no subset $S \subseteq A$ where agents in S can reassign houses among themselves to make all of them strictly better off.

2) Algorithm Description: The TTC algorithm operates iteratively by identifying and executing trading cycles based on agents' top preferences. The key steps are as follows:

Initialization:

- **Active Agents**: $A' \leftarrow A$.
- **Active Houses**: $H' \leftarrow H$.
- **Ownership Mapping**: Each house h_i is initially owned by agent a_i.

Iterative Process:

1) Preference Pointing:
   - Each active agent $a_i \in A'$ points to their most preferred house $h_j \in H'$.
   - Each active house $h_j \in H'$ points to its current owner a_k.
2) Cycle Identification:
   - Construct a directed graph G where nodes represent agents and houses.
   - Edges represent agents pointing to houses and houses pointing to their owners.
   - **Lemma**: Due to strict preferences and finiteness, at least one cycle exists in G.
3) Trade Execution:
   - For each identified cycle:
     - Agents receive the house they point to.
     - Remove these agents and houses from A' and H'.

Termination:

- Repeat the iterative process until A' is empty.
- The final allocation μ is established.

3) Illustrative Example: Consider four agents and four houses:

- **Agents**: $A = \{a_1, a_2, a_3, a_4\}$.
- **Houses**: $H = \{h_1, h_2, h_3, h_4\}$.
- **Initial Ownership**: $owner(h_i) = a_i$ for $i = 1, 2, 3, 4$.
- **Preferences**:

$$a_1 : h_2 \succ h_3 \succ h_1 \succ h_4,$$
$$a_2 : h_3 \succ h_4 \succ h_2 \succ h_1,$$
$$a_3 : h_1 \succ h_2 \succ h_3 \succ h_4,$$
$$a_4 : h_4 \succ h_1 \succ h_2 \succ h_3.$$

a) Round 1:

- **Agents Point to Houses**:

$$a_1 \to h_2,$$
$$a_2 \to h_3,$$
$$a_3 \to h_1,$$
$$a_4 \to h_4.$$

- **Houses Point to Owners**:

$$h_1 \to a_1,$$
$$h_2 \to a_2,$$
$$h_3 \to a_3,$$
$$h_4 \to a_4.$$

- **Cycle Identification**:
  - **Cycle 1**: $a_1 \to h_2 \to a_2 \to h_3 \to a_3 \to h_1 \to a_1$.
  - **Cycle 2**: $a_4 \to h_4 \to a_4$ (self-loop).
- **Trade Execution**:
  - **Cycle 1**:
    - a_1 receives h_2.
    - a_2 receives h_3.
    - a_3 receives h_1.
  - **Cycle 2**:
    - a_4 retains h_4.
- **Update Active Sets**:
  - Remove a_1, a_2, a_3, a_4 from A'.
  - Remove h_1, h_2, h_3, h_4 from H'.

b) Termination:

- Since A' is empty, the algorithm terminates.

c) Final Allocation::

$$\mu(a_1) = h_2,$$
$$\mu(a_2) = h_3,$$
$$\mu(a_3) = h_1,$$
$$\mu(a_4) = h_4.$$

This allocation is both Pareto optimal and in the core, as no agent can be made better off without making another worse off, and no group of agents can reallocate among themselves for mutual benefit.

4) Theoretical Properties: The TTC algorithm ensures:

- **Pareto Optimality**: Agents receive the most preferred feasible house, precluding any reallocation that could make someone better off without harming others.
- **Core Stability**: No subset of agents can deviate to achieve a strictly better allocation among themselves.
- **Strategy-Proofness**: Agents maximize their outcomes by truthfully revealing their preferences, as misrepresentation cannot yield a better result.

B. Cycle Detection Methods

Efficient cycle detection is critical for implementing TTC in a distributed system. We explore two methods: the **Las Vegas algorithm** and a deterministic algorithm by Zheng and Garg [13].

1) Las Vegas Algorithm for Cycle Detection: The Las Vegas algorithm employs randomness to detect cycles in directed graphs.

a) Algorithm Outline:

1) **Token Initiation**: Each agent sends a token along its outgoing edge.
2) **Random Walk**:
   - Tokens carry unique identifiers and hop counts.
   - Tokens move randomly along outgoing edges.
3) **Cycle Detection**:
   - If a token returns to its originator, a cycle is detected.
4) **Trade Execution**:
   - Agents in the detected cycle execute the trade.

b) Characteristics:

- **Advantages**:
  - Simplicity and ease of implementation.
  - Suitable for small-scale or less critical systems.
- **Disadvantages**:
  - Non-deterministic execution time.
  - High communication overhead due to randomness.
  - Less efficient in large or resource-constrained networks.

2) Deterministic Algorithm for Cycle Detection: The deterministic algorithm offers a reliable and predictable approach to cycle detection.

a) Algorithm Outline:

1) **Successor Pointers**:
   - Each agent a_i points to its successor, the owner of its most preferred house.
2) **Pointer Jumping**:
   - Agents iteratively update their pointers to point to the successor's successor.
   - Formally, in iteration k, a_i updates its pointer to $p_i^{(k)} = p_{p_i^{(k-1)}}^{(k-1)}$.
3) **Cycle Detection**:
   - The process continues until pointers stabilize, indicating cycles when $p_i^{(k)} = p_i^{(k-1)}$.
4) **Trade Execution**:
   - Agents whose pointers form cycles execute trades simultaneously.

b) Advantages:

- **Determinism**: Predictable execution time and behavior.
- **Efficiency**: Reduced communication rounds, typically $O(\log n)$ iterations.
- **Scalability**: Performs well in large-scale distributed systems.

3) Integration into Distributed TTC: Incorporating the deterministic algorithm into TTC enhances:

- **Consistency**: Uniform cycle detection across all agents.
- **Reliability**: Elimination of randomness leads to predictable outcomes.
- **Communication Efficiency**: Lower overhead compared to randomized methods.

C. Challenges in Distributed Implementation

Implementing TTC with deterministic cycle detection in a distributed environment poses challenges:

- **Asynchronous Communication**: Variability in message delivery times can lead to inconsistent states.
- **Fault Tolerance**: The system must handle agent failures and message losses gracefully.
- **Synchronization**: Coordinating actions without a central coordinator requires robust protocols.

D. Implementation Considerations

To overcome these challenges:

- **Unique Identifiers**: Assign unique IDs to agents and messages to prevent ambiguity.
- **Logical Clocks**: Utilize logical timestamps (e.g., Lamport clocks) to sequence events.
- **Reliable Communication**: Implement acknowledgment and retransmission mechanisms.
- **State Management**: Agents maintain comprehensive state information to detect and recover from inconsistencies.
- **Consensus Protocols**: Use distributed consensus algorithms (e.g., Paxos, Raft) to agree on cycle formations and trade executions.

III. Implementation of the Distributed TTC Algorithm

Our distributed TTC algorithm was implemented using C++ and socket programming, where each agent operates independently as a networked node. This section details the primary components of our implementation, focusing on the initialization, cycle detection, and trade execution phases.

A. Network Initialization and Connection Setup

Each agent (node) in the system initializes a server socket to listen for incoming connections and establishes client connections with other agents based on their ID to form a fully connected network. This structure enables synchronous and asynchronous communication for message handling and cycle detection.

```
Function setupServer()
    server_fd <- Create a new socket (AF_INET,
        SOCK_STREAM)
    Bind the socket to a specific address and port
    Start listening for incoming connections with a
        backlog size of 10
End Function

Function connectTo(target_id)
    target_address <- Resolve the address of the
        target node using target_id
    socket_fd <- Create a new socket (AF_INET,
        SOCK_STREAM)
    Establish a connection to the target_address
    Store the connection in the connections table
        with target_id as the key
End Function
```

Listing 1. Server setup and client connection

B. Deterministic Cycle Detection

We modified the Las Vegas algorithm to be deterministic in its cycle detection. For this approach, each agent sends tokens (messages) that carry the originator's ID and a hop count, incremented at each step. When a token returns to its origin, a cycle is detected. This implementation ensures determinism by using unique identifiers for both agents and tokens, so cycles are consistently detected across runs.

```
Function handleConnection(socket)
    buffer <- Create a buffer of size BUFFER_SIZE
    While true do
        valread <- Read data from the socket into
            the buffer
        message <- Convert buffer contents (up to
            valread) to a string
        Call processMessage(message)
    End While
End Function

Function processMessage(message)
    (origin_id, hop_count) <- Parse the message to
        extract the origin ID and hop count
    If origin_id = id then
        // The token has returned to its origin; a
            cycle is detected
        inCycle <- true
        Call executeTrade()
    Else
        // Forward the token to the next agent in
            the preference list
        hop_count <- hop_count + 1
        next_agent <- Get the next agent from the
            preference list
        token_message <- Create a token message with
             origin_id and updated hop_count
        Call sendMessage(next_agent, token_message)
    End If
End Function
```

Listing 2. Handling incoming messages for cycle detection

Each agent follows a strict protocol for message forwarding and cycle detection:

- **Cycle Entry**: When an agent receives its own token back, it recognizes a cycle.
- **Token Handling**: Each token is forwarded to the next agent in the preference list, with a hop count to limit potential infinite loops.
- **Determinism**: Using structured message handling and unique identifiers ensures that each cycle is detected predictably, eliminating randomness typical in the Las Vegas algorithm.

C. Trade Execution and Synchronization

Once a cycle is detected, agents within the cycle exchange items. Each agent broadcasts a "trade complete" message to notify others of its updated state, and agents outside the cycle remove those agents and items from their preference lists.

```
Function executeTrade()
    For each agent in cycleAgents do
        Call allocateHouse(agent) // Assign the
            house to the agent
    End For
    Call broadcastTradeCompletion() // Notify all
        agents that the trade is complete
End Function

Function onReceivingCompletion(agent_id)
    Call updatePreferences(agent_id) // Remove the
        completed agent from the active list
End Function
```

Listing 3. Trade execution pseudocode

D. Fault Tolerance and Message Reliability

To guarantee the system operates reliably even with potential message delays or losses, we implemented acknowledgment and retry. Each agent waits for acknowledgment messages from other agents before proceeding to the next step. This ensures that agents maintain a consistent state across the distributed network, with no dependencies on message order.

IV. Results

Results are shown in Fig. 1 and Fig. 2. We compare the result of Total messages versus number of processes, and the result of Execution Time versus number of processes.

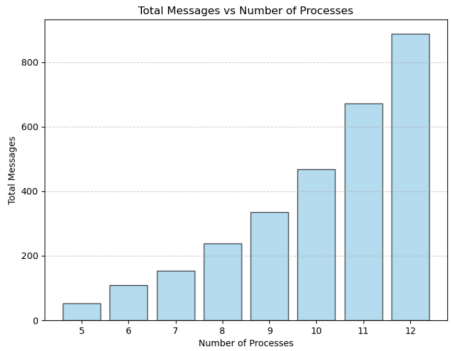

Fig. 1. Total Messages v.s. Number of Processes

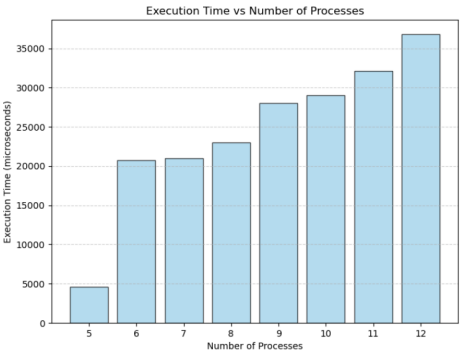

Fig. 2. Execution Time v.s. Number of Processes

Our evaluation focused on the performance of the distributed implementation of the TTC algorithm with deterministic cycle detection. We analyzed two primary metrics: the

total number of messages exchanged and the execution time as the number of agents increased. The results showed that the total messages exchanged grew approximately linearly with the number of agents, reflecting the efficiency of the communication protocol, where messages are forwarded only to relevant agents in the cycle detection process. Execution time exhibited logarithmic growth, demonstrating the algorithm's ability to scale effectively with larger systems. These results highlight the deterministic approach's advantages in reducing redundant communication and computational overhead compared to probabilistic methods.

V. Conclusion

We implemented a distributed version of the TTC algorithm with deterministic cycle detection, addressing asynchronous communication and reliable cycle detection challenges in distributed environments. The implementation demonstrated scalability and efficiency, with manageable communication and computational demands as the number of agents increased. Deterministic cycle detection ensured consistent and predictable results without the overhead of probabilistic randomness, making the approach suitable for decentralized resource allocation applications. However, scalability in highly dynamic environments with frequent agent changes remains a concern, as increased communication overhead could impact performance. The algorithm's reliance on reliable network conditions limits its robustness in high-latency or failure-prone systems, and the lack of privacy-preserving mechanisms may restrict its applicability where agents cannot openly share preferences. Fault tolerance for agent failures is underdeveloped, particularly for disrupted cycles, and further testing is needed under real-world conditions with varying network behaviors and large-scale disruptions. Future work should explore optimizations in message handling, hybrid cycle detection methods, privacy-preserving mechanisms, and recovery protocols for fault tolerance.

Acknowledgment

We would like to express our sincere gratitude to our professor, Dr. Vijay K. Garg, for his invaluable guidance and support throughout this project and class. We are deeply appreciative of the learning opportunities and mentorship he has provided, which have greatly enriched our academic and professional growth.

References

[1] L. S. Shapley and H. Scarf, "On cores and indivisibility," *Journal of Mathematical Economics*, vol. 1, no. 1, pp. 23–37, 1974.

[2] A. E. Roth, T. Sönmez, and M. U. Ünver, "Kidney exchange," *The Quarterly Journal of Economics*, vol. 119, no. 2, pp. 457–488, 2004.

[3] A. Abdulkadiroğlu and T. Sönmez, "School choice: A mechanism design approach," *American Economic Review*, vol. 93, no. 3, pp. 729–747, 2003.

[4] A. E. Roth, "Incentive compatibility in a market with indivisible goods," *Economics Letters*, vol. 9, no. 2, pp. 127–132, 1982.

[5] A. E. Roth, "The evolution of the labor market for medical interns and residents: A case study in game theory," *Journal of Political Economy*, vol. 92, no. 6, pp. 991–1016, 1984.

[6] A. E. Roth and A. Postlewaite, "Weak versus strong domination in a market with indivisible goods," *Journal of Mathematical Economics*, vol. 4, no. 2, pp. 131–137, 1977.

[7] J. Aspnes and O. Waarts, "Randomized consensus in expected $O(n \log n)$ individual work," *Distributed Computing*, vol. 17, no. 2, pp. 115–125, 2005.

[8] S. Fischer and M. Michael, "Distributed matching markets: Welfare and computational complexity," *Distributed Computing*, vol. 30, no. 4, pp. 245–263, 2017.

[9] T. D. Nguyen, A. K. Bachmann, S. Mao, and T. T. Tran, "Distributed resource allocation for network slicing over licensed and unlicensed bands," in *Proceedings of IEEE INFOCOM*, 2016, pp. 1–9.

[10] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.

[11] L. Moraes, D. R. A. Dos Santos, and R. da Silva Torres, "Distributed energy resources management using a consensus algorithm," *IEEE Transactions on Smart Grid*, vol. 9, no. 6, pp. 6054–6064, 2018.

[12] F. Ullah, F. Al-Turjman, L. Mostarda, and R. Gagliardi, "Applications of artificial intelligence and machine learning in smart cities," *Computer Communications*, vol. 154, pp. 313–323, 2020.

[13] Y. Zheng and V. K. Garg, "A fault-tolerant distributed algorithm for the top trading cycle," in *Proceedings of the 29th International Symposium on Distributed Computing*, 2015.