TCB Minimizing Model of Computation (TMMC)

Naila Bushra

TCB minimizing model of computation (TMMC)

By

Naila Bushra

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2019

TCB minimizing model of computation (TMMC)

By

Naila Bushra

Approved:

Mahalingam Ramkumar
(Major Professor)

Eric Hansen
(Committee Member)

Maxwell Young
(Committee Member)

Tanmay Bhowmik
(Committee Member)

T. J. Jankun-Kelly
(Graduate Coordinator)

Jason M. Keith
Dean
Bagley College of Engineering

Name: Naila Bushra

Date of Degree: December 13, 2019

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Mahalingam Ramkumar

Title of Study: TCB minimizing model of computation (TMMC)

Pages of Study: 151

Candidate for Degree of Doctor of Philosophy

The integrity of information systems is predicated on the integrity of processes that manipulate data. Processes are conventionally executed using the conventional von Neumann (VN) architecture. The VN computation model is plagued by a large trusted computing base (TCB), due to the need to include memory and input/output devices inside the TCB. This situation is becoming increasingly unjustifiable due to the steady addition of complex features such as platform virtualization, hyper-threading, etc. In this research work, we propose a new model of computation - TCB minimizing model of computation (TMMC) - which explicitly seeks to minimize the TCB, viz., hardware and software that need to be trusted to guarantee the integrity of execution of a process. More specifically, in one realization of the model, the TCB can be shrunk to include only a low complexity module; in a second realization, the TCB can be shrunk to include nothing, by executing processes in a blockchain network.

The practical utilization of TMMC using a low complexity trusted module, as well as a blockchain network, is detailed in this research work. The utility of the TMMC model in guaranteeing the integrity of execution of a wide range of useful algorithms (graph algorithms, computational geometric algorithms, NP algorithms, etc.), and complex large-scale processes composed of such algorithms, are investigated.

Key words: Trusted Computing Base (TCB), minimal TCB, trusted execution, blockchain, two-party prover-verifier protocol, Authenticated Data Structure (ADS), Ordered Merkle Tree (OMT)

DEDICATION

I dedicate my dissertation work to four of the most important people in my life, my parents Md. Mazibur Rahman and Maksuda Rahman, my elder sister Nazla Bushra and my husband Mehedi Hasan.

ACKNOWLEDGEMENTS

I want to take this opportunity to thank my Ph.D. advisor Mahalingam Ramkumar. I am fortunate to have such a positive-minded and confident person as my mentor. He has taught me how my research work needs to have a real-life impact to make a difference and helped me to always see the big picture. I enjoyed working with him for the last four years.

I am greatly thankful to my committee members Dr. Eric Hansen, Dr. Maxwell Young, and Dr. Tanmay Bhowmik for their helpful insights. They have helped me to evaluate my work from various point of views. They have provided me valuable suggestions that made the focus of my work more explicit.

I thank all the faculty members and staff of the Computer Science and Engineering Department who contributed towards creating a collaborative environment in our department. I thank my fellow graduate students especially my colleague Naresh Adhikari for keeping me informed where I was not.

I am grateful to my sister Nazla Bushra who has always been a role model for me. She is a true inspiration for me to become an active researcher. Last but not least, I am forever thankful to my husband Mehedi Hasan for being there to support me in every decision I have ever made.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE

TCB: Trusted Computing Base

Minimal TCB: TCB with minimal configuration

ADS: Authenticated Data Structure

OMT: Ordered Merkle Tree

OC: Open Consensus

TH: Trusted Hardware

VO: Verification Object

UDI: Unconstrained Data Item

CDI: Constrained Data Item

VN: von Neumann

IS: Information System

RAM Random Access Memory

CPU: Central Processing Unitf

I/O: Input Output

r**:** Root of an OMT

$(k, v)_{k_n}$**:** Logical representation of OMT leaf

(k, k_n, v)**:** Actual representation of OMT leaf

$(P_1, \ldots, P_n)$**:** Processes in a system/algorithm

$(T_1, \ldots, T_n)$**:** Transactions in a blockchain

$(F_1, \ldots, F_n)$**:** State-transition functions

$(R_1, \ldots, R_n)$**:** Registers that store OMT roots $r_0, \ldots, r_n$

$x \parallel y$**:** x concatenated with y

$k \in R$**:** Key k exists in OMT with root R, $\notin$ is non-existence

$(k, v) \in R$**:** Key k with value v exists in OMT with root R, $\notin$ is non-existence

$(k, v)_{k_n} \in R$**:** The next key of k in R is k_n

CHAPTER I

INTRODUCTION

The utility of any algorithm is ultimately limited by our confidence in the integrity of the platform on which the algorithm is executed. The trusted computing base (TCB) for a computing platform is a minimal set of hardware/software that needs to be trusted, in order to guarantee the correctness of algorithms/processes executed on the platform. Von Neumann (VN) architecture is a model of computation described by John von Neumann in 1945 [23]. In conventional computing platforms based on VN architecture, components like CPU, memory, and I/O devices are included in the TCB. VN architecture is also memory-centric; every step in the execution of processes involves memory access, as the program instruction sets and data are all loaded in the main memory.

The von Neumann model is a simple iterative execution of the following steps:

- Fetching instruction pointed to by a special CPU register (instruction pointer or program counter)
- Decoding the instruction
- Fetching operands from the memory (if necessary)
- Executing the instruction

- Writing the result back to the memory (if necessary)

Memory is vulnerable to attacks and can easily be subjected to unauthorized access. Moreover, process execution may involve a complicated sequence of steps (instructions) that may not be readily understandable by all. In other words, VN platforms expose a substantial attack surface that can be exploited by attackers to compromise the integrity of processes, either for selfish gains or more sinister purposes.

A significant number of *memory-based* attacks such as RAM scraping [56], buffer overflow [33], fileless malware [48], cache-based side-channel attacks [67], DMA malware attacks [62], invasive or semi-invasive attacks [57] etc. prove that security features of existing architectures can be exploited in order to illegitimately read, write and/or update information stored in the system's memory. Therefore, memory cannot be completely trusted in practice. Special classes of malware are designed to infect memory and access all the protected information stored in it. Such malware can be identified only by vigorous scanning and analyzing of the main memory [49].

Few of the recent data breaches involve a RAM scrapping malware that captures data from volatile memory. One of the data breach events that happened in 2013 involve two major companies (Target and Home Depot) affected by this malware. In December 2013, over 40 million credit cards were stolen from nearly 2000 Target stores by accessing data on point of sale (POS) systems memory [51]. Home Depot's point of sale system was also compromised [27] in the same way, by the same type of malware. The use of stolen third-party vendor credentials and RAM scraping malware were the main reasons behind both of these data breaches. Data breaches and compromise of system security are not uncommon even in the most recent years as indicated by the fact that already over 4 billion data has already been exposed in 2019 so far. This indicates how easy it for the attackers to gain unauthorized access to the system and steal private information.

Another major flaw discovered recently by researchers might lead to redesigning the kernel software of the operating system. Vulnerabilities like *Meltdown* [35] and *Spectre* [30] are found in modern CPU hardware, and can possibly result in leaking passwords and sensitive data from our systems. Meltdown is a hardware vulnerability affecting Intel x86 microprocessors, IBM POWER processors, and some ARM-based microprocessors. This scenario allows a process to read all memory illegitimately. Spectre combines legit programs with malicious applications and collects secret information from user's computer memory. Both of these vulnerabilities are massive and comparatively harder to mitigate.

Modern CPUs contain advanced features for faster processing of data and instructions. These performance-enhancing optimization features such as out-of-order execution, virtualization, hyper-threading, use of multiple cores, etc. involve complicated sequences of steps. Modern software is also packed with features to cater for various user needs. This makes it harder to identify undesired and obscure functionality in complex software that can easily be exploited. Also, procedural descriptions of system processes are difficult to scrutinize and understand.

Consequently, including modern complex CPUs in the TCB is dangerous. Including memory in the TCB is even more so. Moreover, conventional information security approaches that attempt to mitigate such vulnerabilities tend to possess multiple security layers, that tend to obscure the internal structure of the system itself, making it hard to even understand how the system works. The complexity also makes it difficult to pinpoint the actual root cause of a problem (in case something goes wrong). Therefore, simplifying of TCB is a necessary step.

Minimal TCB, by definition, is comprised of minimal storage and computational requirements and provides a setting to execute processes inside a trustworthy boundary. The minimization of TCB in most of the cases involves getting rid of the main memory to replace it with a smaller one (a few KB of capacity). Several efforts have sought the minimize the TCB using a memory-less architecture [32] [19] [22]. Hendricks *et al.* [28] mentions that even though the main memory and peripherals are made part of the TCB in many cases, there is no reason to trust them blindly. The typical size of the main memory does not allow examining its contents. A small memory does not allow any standard sized malicious program to make copies of itself in it. It is also not possible for these kinds of malware to be executed within this small memory as the few KB size will not suffice. The proposed TCB minimizing model of computation (TMMC) model is fundamentally a minimal TCB that can execute fixed sequences of logical and cryptographic hash operations with its limited configuration.

1.1 Proposal Overview

We propose TMMC (TCB minimizing model of computation) as an alternative to the conventional model of computation. In the TMMC model, memory and I/O devices are not considered parts of the trusted computing base (TCB). In TMMC, an information system (IS) is seen as a set of *processes* or *algorithms*. Process execution is represented as a set of well-defined atomic state-transition functions (ASF). Execution of these functions causes changes to the state of the process. The goal of the TMMC model is to preserve the integrity of the execution of these processes [53] by guaranteeing the integrity of each state-change function.

Each ASF modifies process *states* like inputs, outputs, and temporary data needed to keep track of the progress of execution of the process. They cause the process to change its state (say, from S_1 to S_2) as a result of some data transformation (say, $x = 0$ to $x = 5$). TMMC maps processes of a system to a set of *permitted* state-transition functions ($f : S_n \rightarrow S_{n+1}$). The state-transition functions are atomic in nature which makes them suitable for independent execution and verification. At the end of correct execution of all the atomic state-transition functions ($T_1, T_2, \ldots, T_n$) the process reaches to its most recent state (S_{n+1}).

The TMMC model is essentially a two-party prover-verifier model, where *provers* have to prove the correctness of algorithm execution to *verifiers*. Provers act as the executors of the processes and possess high computational power and memory. They store all the *data items* of the process and execute atomic state-transition functions. As a result of their complex structure, provers can not be considered trustworthy. Verifiers, on the other hand, are only responsible for verifying the correctness of the atomic state-transitions executed by the provers. They do not store any data item of the process, and may not need to verify every state change. The simple configuration of verifiers implies that it is possible to ensure the trustworthiness of verifiers.

More specifically, TMMC can employ a blockchain network for executing algorithms/ processes, where provers are incentivized users who have a stake in ensuring the integrity of the process execution. They can offer proof of the integrity of any state-change to other users (i.e. regular users) at any time. Execution of any algorithm in TMMC can be seen as a sequence of blockchain transactions, each associated with a well-defined atomic state-change function (ASF). The blockchain consensus mechanism enables a universal consensus on the state of the algorithm after each transaction.

The dynamic states of a process (i.e. existence or non-existence of data items, the content of the data items, etc.) are seen as leaves of an Ordered Merkle Tree (OMT) [44]. The OMT is an Authenticated Data Structure (ADS), which is also an extension of the Merkle hash tree [41]. ADS allows the provers to prove the authenticity of its response to specific queries. The OMT is associated with a cryptographic commitment (the root r of a binary hash tree). In TMMC, this root hash r is used as a commitment to the state of the algorithm. The provers store the entire OMT. Verifiers keep track of only the cryptographic commitment r, to keep track of the process's current state. Provers must prove the correctness of process execution to verifiers. Consider a scenario where the prover was queried for the value v of a particular data item x. The provers are required to send the value v along with a "proof of validity". This proof of validity can be used by the verifiers to confirm the correctness of the response against the root r of the OMT. In the same way, untrusted provers can also provide proof of validity of the correct execution of state-transition functions.

The minimal TCB for TMMC can be implemented using "low-complexity-high-integrity" Trusted Hardware (TH). Alternatively, the TMMC model can utilize a blockchain network where consensus is reached on OMT root r following each state-change.

In the traditional von Neumann (VN) architecture, memory can be accessed in constant time ($\mathcal{O}(1)$). However, in TMMC, where data items are OMT leaves, operations like read, write, update, delete, require $\mathcal{O}(\log_2 N)$ time, where N is the total number of data items (OMT leaves). Thus algorithms, when executed on the TMMC platform, may not run as efficiently (compared to the conventional VN model). This is the trade-off for guaranteed integrity of process execution.

The TMMC model for process execution an alternative to the conventional procedural model for executing processes. This research work hypothesizes that the TMMC model can be used for executing a wide variety of algorithms used in real-world applications. In the proposed research work, we convert processes/algorithms into a sequence of TMMC state-transition functions and evaluate the associated computational overhead. We also analyze the state-transition rules for a wide variety of application scenarios, to formalize their notations.

1.2 Research Goals

The broad goals of this research work are described as follows:

1.2.1 Generic Platform for Trustworthy Execution of Processes

The proposed research seeks a trusted model of computation that can be used for a trustworthy execution of any algorithm processes from any domain of interest. This approach emphasizes preserving the integrity of process execution as opposed to conventional approaches that lay more emphasis on detecting and thwarting attacks that may affect the integrity of process execution. A major component of this research involves the representation of different useful algorithms as a sequence of TMMC state-transition rules, for efficient execution.

1.2.2 Unambiguous Process State Description

In the conventional model of computing, the description of the process state is implicit due to the procedural description of processes. In TMMC, processes are described as a series of atomic state-transition functions (ASF). Execution of an ASF changes the state of the process. Thus, changes in the process state are explicit in TMMC due to the ASF representation of processes.

1.2.3 Reducing Dependency on Conventional Memory

The TMMC model does not rely on the integrity of memory contents of the process executor, as memory is inherently insecure and can be illegitimately modified. Instead, data read and write are performed using ADS [63], viz., the Ordered Merkle Tree (OMT) [9].

1.2.4 Eliminating the Need for Trusted Third Party

In scenarios where there is a need to trust third parties for the execution of processes, we can not ignore the possibility that third parties can themselves be corrupted or exposed to malicious attacks. TMMC model eliminates the need for trusted third parties, by replacing them with untrusted provers. Instead of blindly trusting third parties, TMMC verifiers demand the proof of correctness of process execution.

1.2.5 Lowering Verification Complexity

Performing verification of process execution is not always practical when the computing platform follows the traditional von Neumann model. TMMC seeks to reduce computational overhead and memory requirements for verification. The process description in TMMC allows the verifiers to verify the correctness of process execution without actually executing all the system processes.

1.2.6 Minimizing TCB

In von Neumann architecture, the TCB includes all the major components such as memory, I/O devices, etc. This makes the size of the TCB large with inherently vulnerable components such as memory. TMMC actively seeks to minimize the TCB for process execution (by excluding memory, I/O devices, etc.). More specifically, the TCB for process execution can be shrunk to nothing, by executing processes in a Blockchain network, where no software/hardware is trusted.

1.2.7 Scalable

The data items of a process are stored as leaves of an OMT. An unlimited number of data items N can be stored in an OMT. To verify a particular ASF execution (process state change), the verifier only needs to perform $\mathcal{O}(\log_2 N)$ hash operations. This makes TMMC highly scalable for large-scale processes that involve a large number of data items.

1.3 Summary

In this chapter, we have provided a brief overview of the significance and motivation behind the proposed TMMC model, followed by a brief overview of the model itself. Chapter 2 provides a more in-depth overview of several key areas that were explored to undertake this research. Chapter 3 describes the major components of TMMC in detail and their roles. Chapter 4 provides a generic TMMC workflow that can be used to transform any process and represent them as TMMC ASFs. We then describe several popular algorithms from different domains (Graph algorithms, computational geometry algorithms, NP problems, etc.) using the TMMC model in the following chapters. In the concluding chapter, we highlight the contributions of our research and scope for future work.

CHAPTER II

LITERATURE REVIEW

2.1 Model of Computation

Traditionally, computable processes are executed primarily using the von Neumann (VN) [23] architecture (Figure 2.1), where main memory is assumed to be trusted. In practice, programs and data are subjected to unauthorized modification, as it is possible for other processes and peripherals to access/modify the memory of a process.

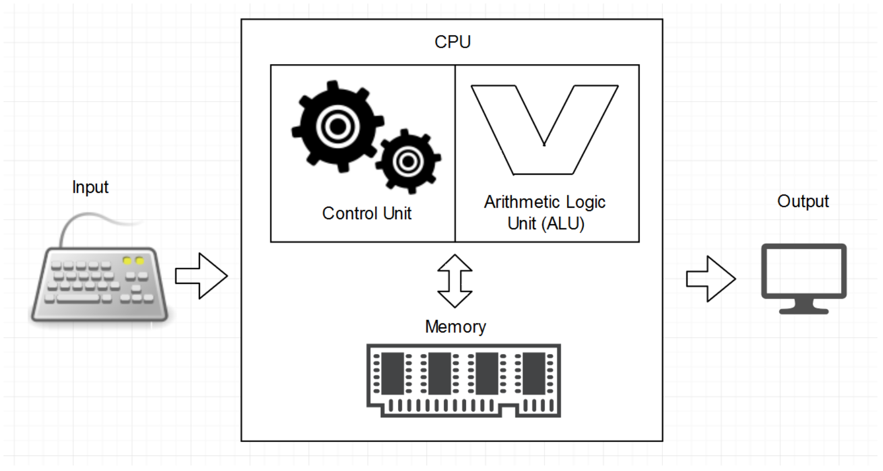


Figure 2.1

von Neumann architecture

The Harvard architecture for computation [60] (Figure 2.2) stores the program code into a separate unit that is read-only. However, data remain in the main memory which is vulnerable to outside attacks. Other similar architectures have also been considered in the literature to render conventional computational models more trustworthy.

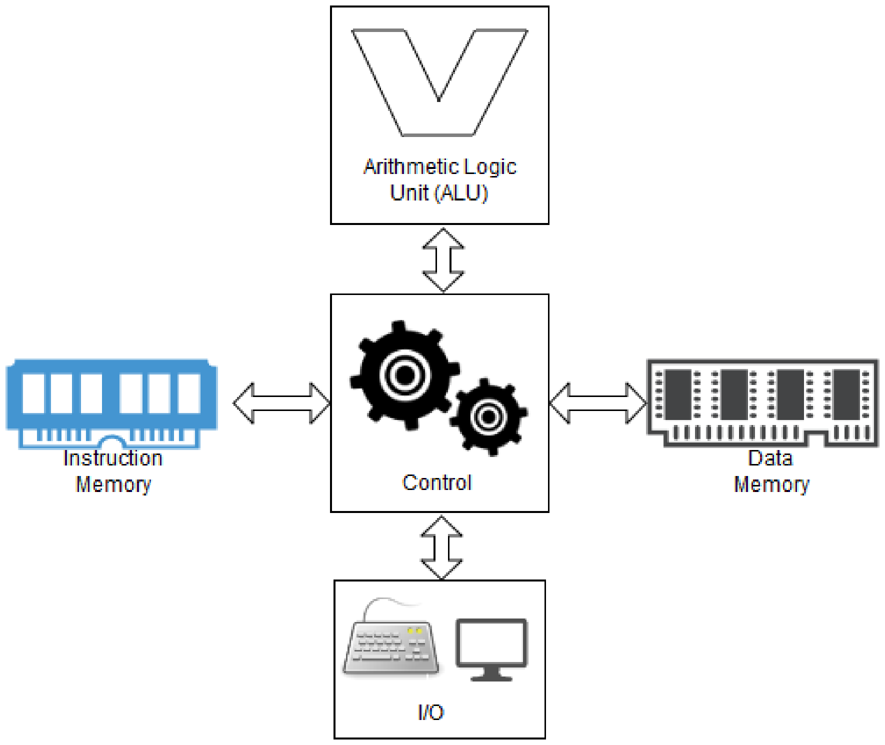


Figure 2.2

Harvard architecture

2.2 Trusted Computing Base (TCB)

The earliest mention of the concept of a trusted computing base was by Rushby [55] where he referred TCB as the combination of kernel and trusted processes. According to Lampson *et al.* [31], the Trusted Computing Base (TCB) of a system is defined as “a small amount of software and hardware we rely on, and that we distinguish from a much larger amount that can misbehave without affecting security.” The Orange Book [34] defines TCB formally as “the totality of protection mechanisms within it, including hardware, firmware, and software, the combination of which is responsible for enforcing a computer security policy."

The integration of the notion of TCB for any system makes it easy to extend the system functionality, as everything outside the TCB can be modified without restrictions. For example, embedded systems are inherently vulnerable to security threats due to network connectivity and third-party extensibility. TCB allows the separation of domains in embedded systems which results in single and well-defined activity for individual components [54]. The notion of a TCB is also used in air traffic control software [17] towards building a larger automated airspace computing system. Research [28] suggests that it is difficult to limit only the processor to be inside the TCB, as to provide a secure boot of the system, other components (e.g. disks, network adapters, etc.) should also be made a part of the TCB. As all components inside the TCB are trusted, any security loophole inside the TCB can cause the insecure behavior of the system. For example, if the TCB that is a part of the Java Compiler (JVM) has some unreliable components, the bugs of the program written by using it can lead to creating security holes in this arrangement [11].

2.2.1 Minimal TCB

Minimizing the TCB [11] [52] [64] is a major field of research. In practice "minimizing the TCB for process execution" implies reducing the computational and memory requirement for the TCB in such a way that it would still be able to handle large-scale data items [68].

There are several broad design principles for minimizing the TCB [26] [40]. For example, minimal TCB can consist of only a part of the application code [40] [39] that is critical to prevent security breaches. The minimization of TCB can be useful for model-driven verification of system states [52] for a variety of systems [51]. One of the major goals of minimizing the TCB is to reduce the verification overhead for a system with dynamic information flow [11], [36]. Minimal TCB is also an integral part of ensuring security in cloud infrastructure [9].

2.3 Two-party Protocol

At the core of all two-party protocols are two entities – *provers* and *verifiers*. The communication between the two parties can be interactive (through message passing) [37] or non-interactive (zero-knowledge proof) [50]. The central idea is that the provers should be able to demonstrate that the information they are providing is valid. Let us suppose, a query q is made to the provers by an entity. The provers should provide a "proof of validity" along with the response r to the entity. The response r together with proof of validity can be used by the verifier to ensure the correctness of response r (Figure 2.3).

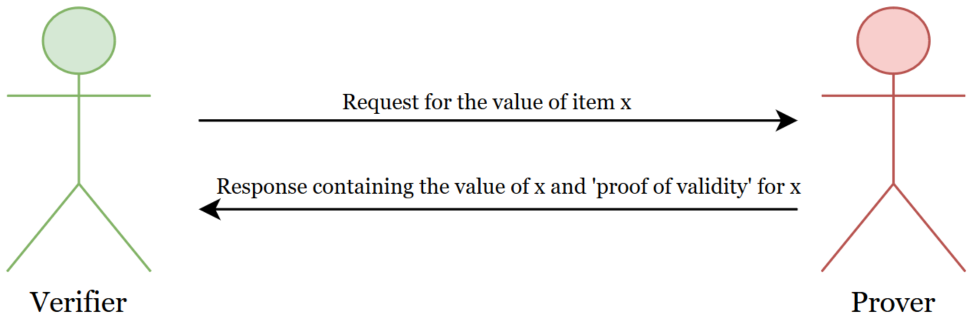

Request for the value of item x
Response containing the value of x and 'proof of validity' for x
Verifier
Prover

Figure 2.3

Two-party protocol

In [47], Parno and Gentry introduce *Pinocchio*, a two-party protocol with *clients* and *workers*. The clients are *verifiers* and the workers are *provers*. The clients usually have low computational resources, and outsource their computation to the untrusted workers. In Pinocchio, the authors have described a model where the clients will be able to verify the correctness of the computation performed by the workers. Pinocchio supports zero-knowledge verifiable computation where the untrusted worker can convince the client of something without revealing it to the client. *Pinocchio*'s proof size is constant and the proof can be generated in linear time. Pinocchio seeks to reduce verification time.

Ben-Sasson *et al.* [7] propose a system that provides succinct non-interactive zero-knowledge proofs (*zk-SNARKs*). In this two-party protocol, *clients* act as *verifiers* and *servers* act as *provers*. The client wants to be sure about the response that was given by the server. This response can be an output of a certain program that is known to both of the clients and the server. Interestingly, this model addresses the assurance from both sides. The server also maintains confidentially of the database and only shares information necessary to answer the query, without revealing the entire database to the client.

2.4 Authenticated Data Structure (ADS)

The Authenticated Data Structure (ADS) [63] is a good fit for any two-party prover-verifier protocol. In a client-server paradigm, the server is referred to as the prover and the client is referred to the verifier. The untrusted prover is responsible for storing and maintaining the ADS. The prover stores the data items of the system as a set of *records* within the ADS. Upon a query from the verifier, the prover is responsible for sending responses to these queries along with a proof of validity. ADSs are structured in such a way that it is possible to obtain a *cryptographic commitment* for the set of *records* stored by it. The verifier only stores this cryptographic commitment and checks the correctness of the response of the prover against it.

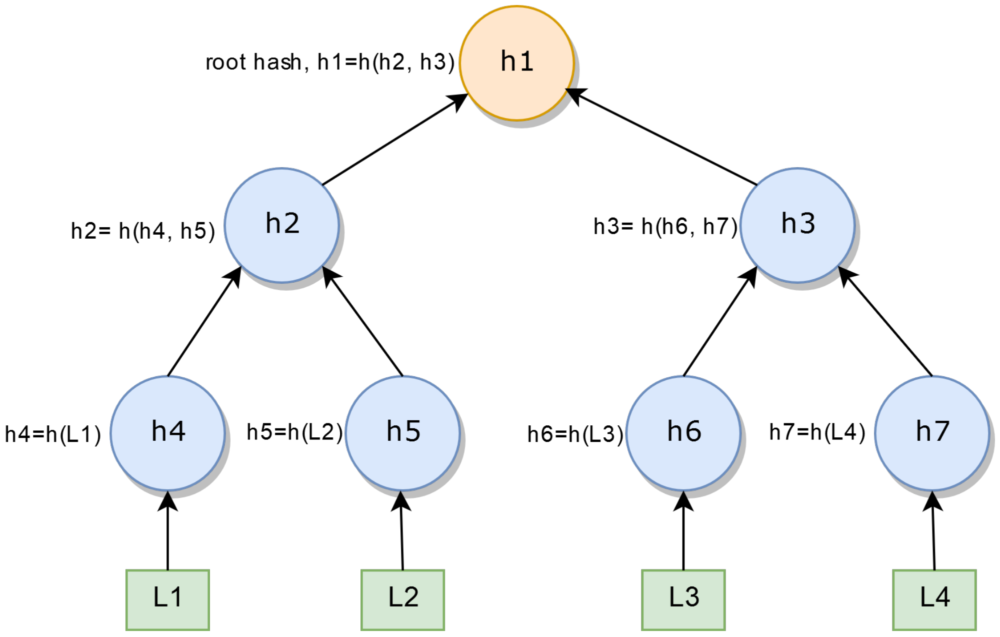


Figure 2.4

Merkle hash tree

Merkle hash tree [41] (Figure 2.4) is a very well recognized example of ADS. Specifically, the Merkle hash tree is a type of ADS that is built using standard cryptographic one-way hash functions $h()$. In Merkle tree leaf nodes are hashes of data blocks (L_i) and non-leaf nodes are hashes of child nodes (h_i). Checking the existence of a leaf node in a Merkle tree with N leaves requires $\mathcal{O}(\log_2 N)$ hash operations.

The potential of ADSes have been well recognized in the literature. ADSes are also widely used in distributed ledger systems like blockchain networks. Many different types of ADSes have been constructed for diverse applications such as skip-lists [61], red-black trees [2], B-trees [18], Merkle Patricia Tree [69] (used in Ethereum cryptocurrency), network management systems, and geographic information systems (GIS) [24], database outsourcing [46], search DAGs [38], etc. An interesting work by Miller *et al.* [43] involves developing a programming language that can be used to transform any existing data structure to authenticated ones such as authenticated binary search trees, authenticated red-black trees, authenticated skip lists, etc.

2.5 Blockchain Network

Blockchain network also utilizes the prover and verifier model and can provide assurance on the integrity of a system through consensus. This approach of open consensus is applied in many popular cryptocurrencies (e.g. Bitcoin [45], Ethereum [69], Moero [65], Leo [3], Zerocoin [42], libra [6] etc.)

A blockchain network is a broadcast network where peer-to-peer communication is used as a mechanism for broadcasting. Every participant in the blockchain maintains a copy of the blockchain ledger. A blockchain ledger is "not" a distributed ledger; every participant maintains the whole ledger. The goal of a blockchain network is to achieve a universal consensus on the correctness of all entries in the ledger. Specifically, explicit consensus on the hash of the ledger is an implicit consent on the entire ledger.

A blockchain is a sequence of blocks that are chained together. Typically ledger updates happen a block at a time The number of blocks grows with time. The blockchain network (Figure 2.5) is

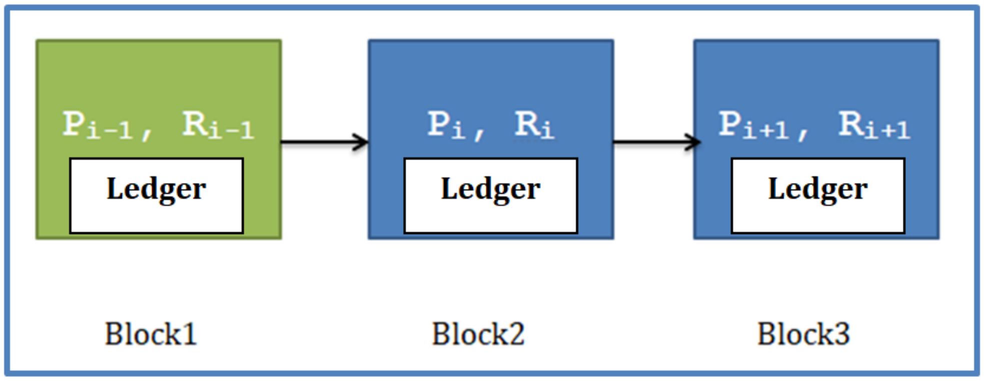


Figure 2.5

Sample blockchain

a *decentralized* network where every participant of the network has a copy of the ledger i.e. the list of transactions. P_n denotes a sequence of transactions. R_m is a cryptographic hash denoting the system state. R_i is the current system state which is acquired by executing P_i on the previous system state R_{i-1}

The contents of a block in a blockchain are primarily the cryptographic hash of that block, the previous block, and a set of transactions. Each block contains a public list of *transactions* or *records*. Each record in the ledger is a well-formed transaction (broadcast over the network). Multiple consecutive transactions are grouped into a block.

Adding a block is only possible when there is agreement on the correctness of all records/transactions in the block, through a consensus mechanism. Miners are blockchain network participants that typically compete amongst themselves to decide "who is going to make changes (add a block)" to the blockchain. Typically, miners solve puzzles to provide "proof-of-work" to be eligible to make changes to the blockchain. Once the change is made, all participants update their ledger.

Any participant of the network can execute all the transactions in the ledger at any time to verify the correctness of every ledger entry.

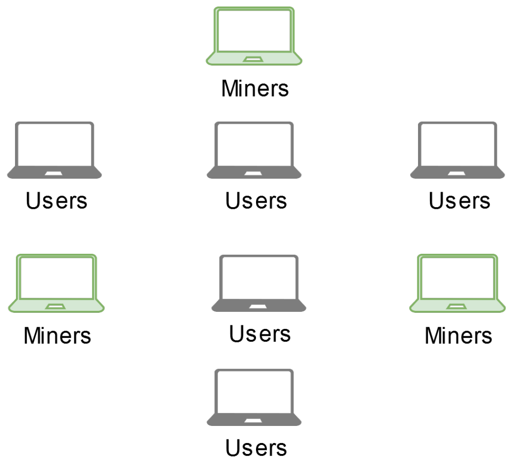


Figure 2.6

Blockchain broadcast network

Blockchain networks are used in many application domains. Most approaches based on blockchain networks [45] [13] [69] attempt to guarantee the integrity of process execution through open consensus. Even while the assumption at the core of open consensus is that “anyone can audit the blockchain ledger at any time” there has been very little focus on several practical issues. For example, some blockchain-based approaches require going through the entire transaction history in the ledger to perform effective audits. For instance, to audit the bitcoin blockchain ledger, one has to download all the transaction records (all the ledger entries) that ever took place from the first block (also known as *genesis* block) to track or verify the current state of the system represented by the latest block. This makes it very difficult for participants with limited resources to verify the system state. In practice, only a special class of entities (miners) perform this comprehensive audit as this mechanism is computationally expensive. In Ethereum [69] the system states are represented using the leaves of a hash tree whose root is explicitly included in transactions that are sealed to the blocks. Consequently, the current state of the system is more readily accessible. Ethereum also aims to minimize the overhead of the process executor (prover). It stores its system states in an ADS – the Merkle Patricia Tree [69]. This makes it possible for users to selectively verify the correctness of any Ethereum transaction. Specifically, verifying the correctness of any Ethereum transaction involves executing a "smart-contract" on an Ethereum virtual machine (EVM). However, there are no strict limitations of the size of the smart-contract or the scale of data to be handled by each contract, or even the work-bench memory used by the EVM (which is an infinitely expandable byte-array).

CHAPTER III

COMPONENTS OF TMMC MODEL

With the combination of well-defined cryptographic protocols and a well defined TCB, TMMC provides a trusted platform to execute any process. This chapter describes the components of the TMMC model.

3.1 Processes in TMMC

In the increasingly digital world, the integrity of dynamic bits that describe states of a wide variety of information systems is crucial. Processes within these systems are responsible for reliably and correctly changing data items representing the system, which can be seen as a set of finite states. The integrity of these data items is critical to ensure the correctness of the system state.

It is important to note that our focus is to ensure the correct execution of a given algorithm. We do not care how or where an algorithm was executed, as long as one can verify the correctness of the solution. In general for any algorithm $o = f(i)$ that produces an output o (possibly a vector of states) for an input i (a vector of input states), there exists a verification algorithm $V(f(), i, o)$ that outputs a binary decision (TRUE/FALSE). At worst, the verification algorithm $V()$ is only as complex as $f()$ (as we can simply execute $o = f(i)$ to verify correctness. Generally, the verification algorithm is simpler. For algorithms in class NP, verification can be substantially simpler. TMMC processes merely need to execute verification algorithms. More specifically, provers merely need to prove the correctness of a sequence of ASFs corresponding to verification algorithms; verifiers merely need the ability to selectively verify the correctness of any ASF.

3.2 Two-party Protocol

Consider a scenario where an algorithm of complexity $\mathcal{O}(N^r)$ needs to be executed to produce an output O, and that everyone needs to be convinced of the correctness of O. For large N, it is obviously impractical for everyone to execute the algorithm themselves. In such scenarios all users may have no choice but to rely on "trusted third parties" to perform $\mathcal{O}(N^r)$ work to execute the algorithm on their behalf. The obvious disadvantage of such an approach is the lack of a rationale for the trust in the third party.

The TMMC model of computation seeks to eliminate the need for third parties. Instead of a trusted third party, TMMC utilizes untrusted provers. Instead of blindly accepting the validity of the information provided by the third party, TMMC verifiers demand proof of correctness. Under the TMMC model of computation, the work done by the prover for executing a $\mathcal{O}(N^r)$ complexity algorithm, and generating a proof of correctness, will range from $\mathcal{O}(N^r)$ to $\mathcal{O}(N^r \log N)$. The work done by verifiers is typically $\mathcal{O}(\log N)$. Provers execute the system processes as atomic state-transition functions and need to prove the correctness of execution to the verifiers.

The prover in the two-party protocol followed by TMMC is responsible for storing data items as leaves of an Ordered Merkle Tree (OMT) [44]. It is important to note that the prover is not a trusted entity as its memory can be illegitimately accessed, or the prover itself can be malicious. The verifier is an entity in the two-party protocol who is responsible for verifying the correctness of the process execution performed by the prover. In the TMMC scenario where the verifier is a low complexity trustworthy module, the module is assumed to be universally trusted. In the TMMC scenario where a verifier is a participant in a blockchain network, the verifiers are merely assumed to trust themselves (and any device they use to verify the simple "proof of correctness" submitted by provers).

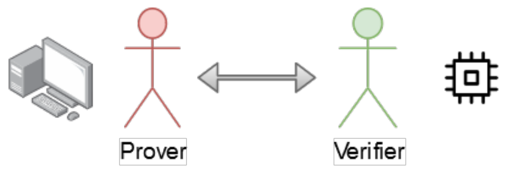
Data items subjected to change as a result of process execution are stored as leaves of the OMT. The location where prover digitally stores the OMT structure is irrelevant. In general, prover possesses high computational power, resources, and unlimited memory as they need to execute the system. The verifier stores the single cryptographic commitment to the OMT. The *root hash* of the OMT is considered as a *cryptographic commitment* in our setting. The root hash is computed by successively hashing the nodes of the OMT at each level starting from the leaves until we reach the root node.

For verification, the verifiers are given a "proof of validity" along with some verification objects (VO) that greatly reduces the overhead of verification for the verifiers. An algorithm that may perform efficiently under the von Neumann model may not be efficient under the TMMC model. This is because while memory read/write in the von Neumann architecture takes $\mathcal{O}(1)$ time, it takes $\mathcal{O}(\log N)$ in the TMMC model.

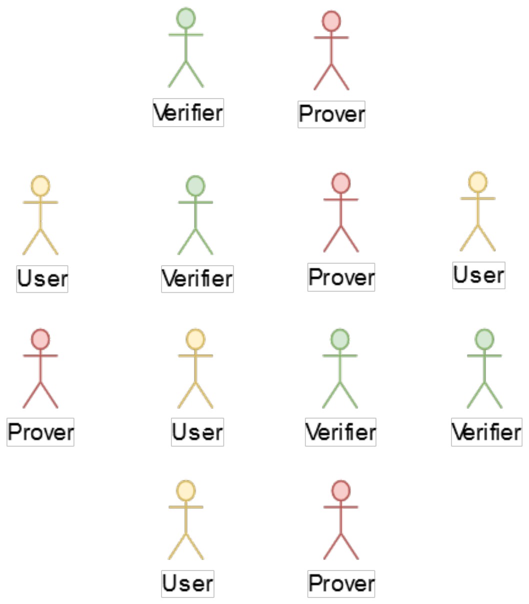
There are two practical platforms where TMMC model can be utilized:

1. Using trusted hardware module, or

2. Using a blockchain network.



(a) TMMC using trusted hardware (TH), Untrusted provers execute the state-transitions and possesses high computational power and memory. Verifiers are comprised of Trusted Hardware (TH) and possesses low computational power and memory



(b) TMMC using Open Consensus (OC). Provers, verifiers and users are parts of the blockchain broadcast network. Provers execute state-transitions in the blockchain. Anyone (verifier or user) can verify the system's correctness at any given time.

Figure 3.1

Utilization of TMMC using trusted hardware and blockchain network

3.2.1 TMMC Trusted Hardware Module

In this approach, TCB has only one component - a high-integrity-low-complexity trusted hardware module (or trusted chip), which acts as the verifier. Ideally, such a hardware module should possess very low computational power and be read-write proof. It should not include any complex units (e.g MMU, cache, functional OS, network hardware, advanced I/O, etc.). The module should be kept isolated physically and digitally to prevent unauthorized access. Its memory should also be small (a few kilobytes), and should not depend on the scale of the process data items.

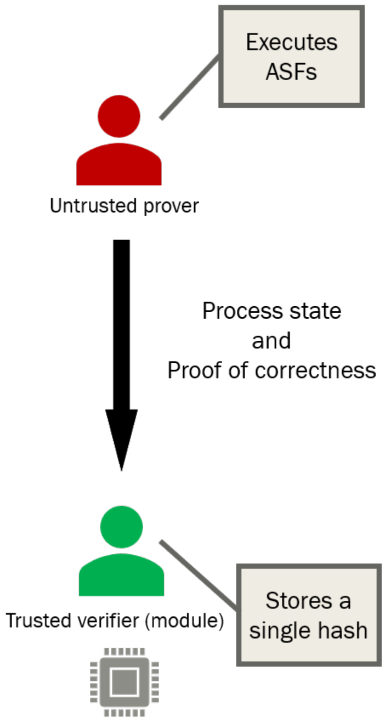

Executes ASFs
Untrusted prover
Process state and Proof of correctness
Trusted verifier (module)
Stores a single hash

Figure 3.2

Prover-verifier communication in trusted hardware setting

Provers can be any untrusted entity outside the module. These untrusted provers execute the atomic state-transition functions (Figure 3.2). They provide the next process state along with the "proof of correctness" (to the verifier). A trusted verifier virtually store data items meaning it only stores a single hash (commitment to current process state). The trust in verifiers is validated by their minimal configuration. Verifiers merely need to store a dynamic cryptographic hash (the root of a binary Merkle tree). They execute $\mathcal{O}(\log_2 N)$ cryptographic hash operations to validate proof of correctness (given by the prover). On successful verification, verifier updates hash (commitment to process state) as suggested by the prover. TMMC model using trusted verifiers is especially useful in scenarios where some information may need to be kept private.

3.2.2 Blockchain-TMMC Network

A blockchain network is a second practical way of utilizing the proposed TMMC model. In blockchain-TMMC, no software or hardware is trusted. A blockchain broadcast network is a mechanism for achieving universal consensus on the correctness of all entries in a blockchain ledger. Specifically, an explicit consensus is reached on the cryptographic hash of the entire ledger, which is an implicit consensus on every ledger entry. Each ledger entry corresponds to a blockchain transaction, broadcast over the network. In blockchain-TMMC, only assumptions are that the ledger is unalterable and every ledger entry is correct (due to trust in the consensus mechanism).

TMMC processes are represented as a sequence of ASFs. In blockchain-TMMC, a process is a sequence of transactions. Each transaction is a trigger for a well-defined ASF that modifies the state of the process executed on the blockchain network.

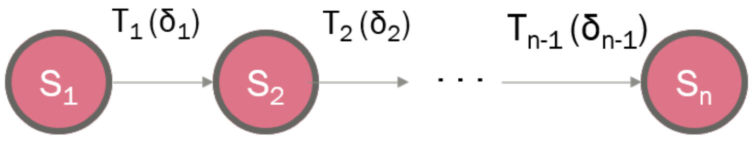


Figure 3.3

Process state change in blockchain-TMMC

A blockchain ledger is a record of the progression of process state S_1 (Figure 3.3), that can be interpreted as a finite state machine (FSM)

- δ_1: $T_1 \times S_1 \rightarrow S_2$
- Transaction T_1 triggers the change of the state S_1
- δ_1 is the well-formed function that changes the system state S_1 to S_2

Each of the blocks in a ledger is linked together. The number of blocks grows with time. The most recent block represents the current state of the system. The contents of a block include a cryptographic hash (R_i) for sequence of transactions for a process (P_i), and the previous block hash R_{i-1} (Figure 3.4). R_i is the current state of the system after executing process P_i on R_{i-1}.

Participants (Figure 3.5) in a blockchain network can be categorized into i) incentivized users (provers) who participate at all times, and ii) regular users (verifiers) who may participate sporadically. In blockchain-TMMC, incentivized users do not need to perform unnecessary “proof of work” in order to be selected to make a motion. Following each transaction, an incentivized user is either i) randomly selected, or ii) compete to be selected and makes a non-repudiable motion to include the “well-formed” transaction in the ledger, or ignore a transaction as ill-formed. Only well-formed transactions are added to the ledger. More often, such motions are made for a set

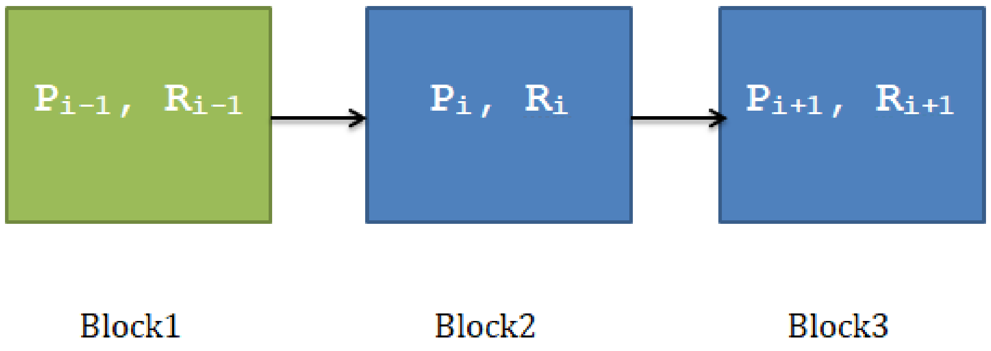


Figure 3.4

Blockchain Network

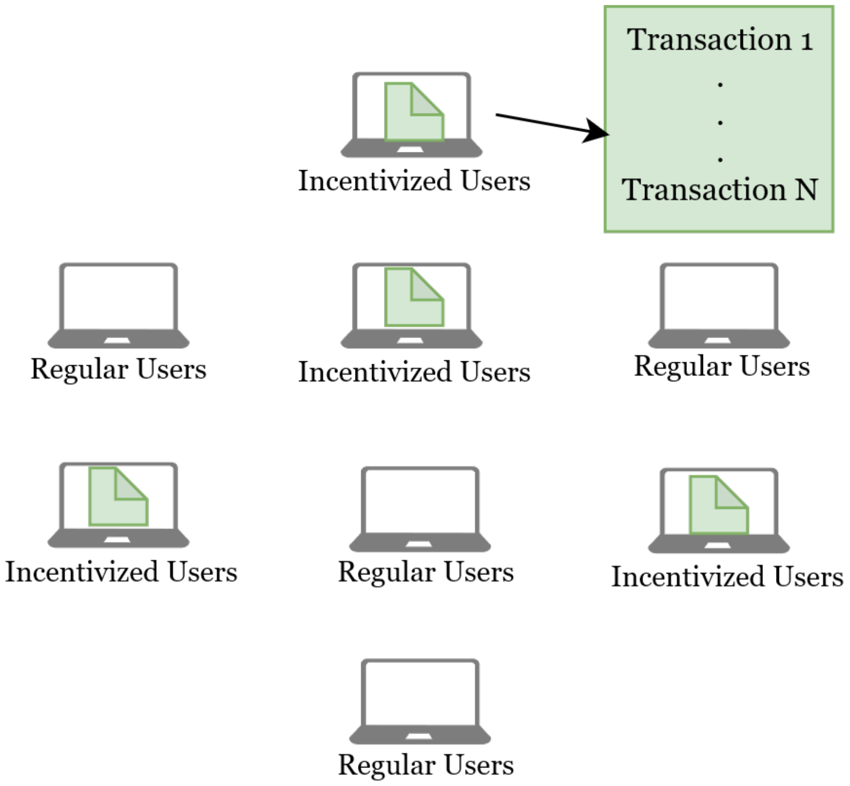


Figure 3.5

TMMC blockchain broadcast network

of transactions to be included in the next block of the ledger. The goal of the incentive mechanism is to reward (incentivized) users for correct motions and punish them for incorrect motions. The participation of regular users becomes mandatory only if a motion by an incentivized user is challenged.

The blockchain platform is the preferred implementation of the TMMC model. Here, TMMC algorithms are executed in a blockchain network and the incentivized users instead of wasting their resources on solving useless puzzles, they perform useful work in order to minimize the work that needs to be done by verifiers (other users).

More generally, the consensus need not be merely on the ledger hash. It can be reached on any computable value. In the blockchain-TMMC, incentivized users are the provers. The consensus is reached on the state of the process after every transaction.

Consider a scenario (Figure 3.6) where universal consensus exists on the state S_n of the process before a transaction T_{n+1}, and that a motion was made by an incentivized user is to update process state to S_{n+1}. Assume that another incentivized user challenges the motion and claims that the new state should be S'_{n+1} instead. Under such scenarios incentivized users are expected to broadcast proof of correctness of state-change $f(S_n, T_{n+1}) = S_{n+1}$ or $f(S_n, T_{n+1}) = S'_{n+1}$. The participation of regular users is necessary only under such scenarios. It is up to each user to verify the two "proofs": one for $f(S_n, T_{n+1}) = S_{n+1}$, and the other for $f(S_n, T_{n+1}) = S'_{n+1}$. As long as state-change functions are simple and unambiguous, there can be one correct next state - at least one of the proofs should be wrong. In TMMC, only $\mathcal{O}(\log_2 N)$ effort is required for users to verify a proof.

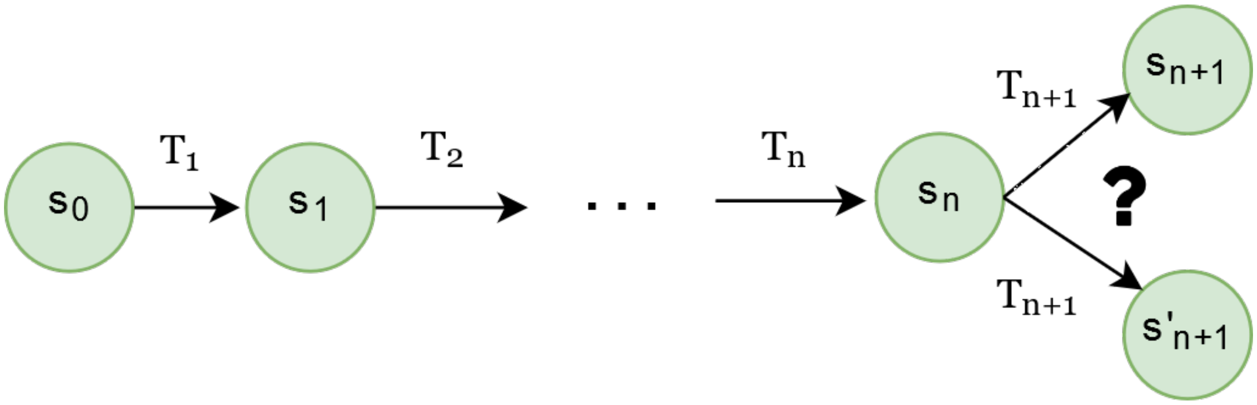


Figure 3.6

Transactions in a blockchain-TMMC network

Blockchain-TMMC platform is open to the public, it is possible to integrate TMMC using blockchain network along with trusted hardware module to execute private processes. This can be achieved by handling the private information of the database in trusted hardware and the rest using the blockchain network. A suitable mapping should exist between the two-parts, which is only known to the trusted hardware module.

3.3 Authenticated Data Structure

3.3.1 Data Items

Any information system can be seen as data-items, and rules for modifying data-items. Modification of data changes the state of the system. In the TMMC model, we recognize data items of two types:

1. Unconstrained Data Items (UDI)

2. Constrained Data Items (CDI)

UDIs are the input to the state-transition functions (they act as triggers). There is no restriction on how this type of data item as it is not feasible to restrict the type of trigger sent by anyone. CDIs, on the other hand, are data items internal to the system. Alternation of CDIs causes changes to the system states. There is a specific and predefined rule (Transformation Procedure, TP) about how CDIs should be modified within the system. These CDIs are stored as leaves of the Ordered Merkle Tree (OMT). These leaves are actually *key-value* pair i.e. (k, v) where k is the unique identifier and v is the value of that CDI.

3.3.2 Ordered Merkle Tree (OMT)

The Merkle Hash Tree [41] is an Authenticated Data Structure (ADS) [63] which is built using standard cryptographic one-way hash functions $h()$ (e.g SHA-1). Merkle tree can answer existence queries of data items. Ordered Merkle Tree (OMT) [44] is a flexible extension of the Merkle Hash Tree that can also answer nonexistence queries.

Authentic Data Structures (ADS) is used to obtain a cryptographic commitment for a set of *records*. The TMMC model uses the OMT to store the data items of a system. In the TMMC model, the cryptographic commitment is the hash value of the "root" of the OMT. Provers store the data items of the algorithm in this OMT. The verifiers only store the cryptographic commitment i.e. root hash in its memory. Thus, the overhead of verification for the verifier is reduced.

The data items of the algorithm are stored as leaves of the OMT and are logically represented as a set of *key-value* pairs i.e. (k, v); the entire OMT is a collection of key-value pairs. The actual organization of the key-value pair can be represented as (k, k_n, v) where k_n is the next key (for key k). OMT collections can be dynamic as new key-value pairs can be added, and existing key-value pairs can be deleted. These collections can be nested as well i.e the value in a key-value pair can be an OMT root that logically represents another OMT collection. The root hash of the top-most collection is the main cryptographic commitment stored by the verifier.

In the conventional memory-centric model reading and writing from the memory take $\mathcal{O}(1)$ time. However, in the TMMC model where the memory operations are performed in OMT, all the basic operations i.e read / insertion/ update/ deletion take $\mathcal{O}(\log_2 N)$ time where N is the number of leaves. The structure of the OMT allows the keys of data items to be in the sorted order inherently. For example: $k_1 > k_2 > k_3$ or $k_1 < k_2 < k_3$. This means k_2 is the next key of k_1, k_3 is the next key of k_2 and k_1 is the next key for k_3. As a result of this inherent structure operation such as getting of extracting the minimum or maximum value from the OMT is also $\mathcal{O}(\log_2 N)$. Accessing data items within OMT can be compared to searching the item as we need to traverse through the OMT to access a data item.

In Figure 3.7 we see a tree with a height of $H = \log_2 N$ (no of leaves, $N = 4$). The lower level (level 2) of this binary hash tree has the corresponding hashes (h_4, h_5, h_6, h_7) of the 4 leaves (L_1, L_2, L_3, L_4). The hashes of level 1 are computed by hashing together a pair of sibling nodes of level 2. This means nodes at level 2 e.g. $h_4 = H(L_1)$. This means two sibling nodes at the same level are hashed together using $h()$ to compute their common parent's hash. This is continued until the top level of the tree is reached. Given the value of leaf L_2, we can calculate its hash h_5. h_5's complementary hashes (h_4, h_3) can be used to recompute the root hash h_1. This is useful to learn about the existence or non-existence of the data item (L_2) in an OMT with root h_1.

A particular data item can only be changed if it exists in the OMT. These complementary nodes or Verification Objects (VO) of these data items are given to the verifier by the prover to check that.

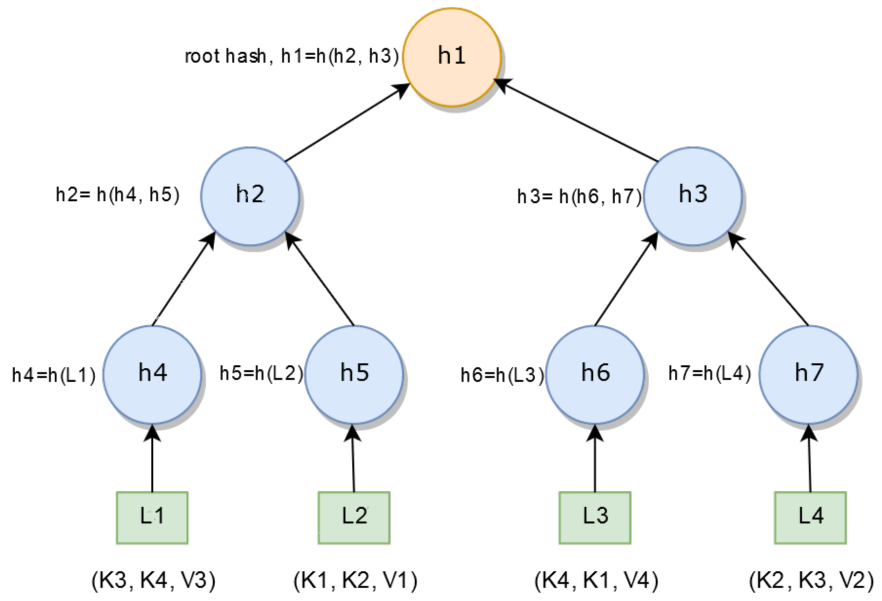


Figure 3.7

An ordered Merkle hash tree with leaves $L_1, \ldots, L_4$

Example: $h_1, \ldots, h_k$ are the complementary hashes for the account key x. The verifier calculates the root r_0 where $x \in r_0$ and compares it with its stored root r. If they are the same then $x \in r$, else $x \notin r$. Non-existence in an OMT has a special representation. Non-existence of item x in OMT R is shown as either $x \notin R$ or $(x, 0) \in R$. This means if we want to delete the item x from OMT R, all we have to do is set its value to 0 i.e. $x \leftarrow 0$.

The numbers of OMTs to be used to store the data items of the algorithm at hand are fixed. The structures of the OMT are also pre-defined to suit the need of the type of data items. For example, while dealing with graph algorithms, we represent input graphs (nodes and edges) using a key-value pair. Keys can be the graph nodes and the connected edges can be the values. For computational geometry problems, many geometrical shapes such as points, lines, triangles are represented as key-value pairs (with coordinates) in the OMT. The root hashes of these OMTs are stored by the verifiers in their fixed number of registers. This memory constraint for a particular process execution allows TMMC to eliminate illicit memory access threats.

Several memory-constrained and *constant-work-space* algorithms are proposed for various problems that need to be executed in a confined execution environment. Asano *et al.* proposed several constant-work-space algorithms for geometrical problems in [5] [4] where they are executed in a resource-limited setting where memory cannot be accessed in constant time. They showed their proposed way of structuring their data items allows them to access them in linear time in a constant-work-space environment. TMMC, however, is a resource-limited environment that takes logarithmic ($\mathcal{O}(\log_2 N)$) to access its data items from OMT.

For large-scale applications, OMT can store unlimited data items as its leaves. The size of the data items does not have any effect on the verification complexity of TMMC as most of the verification boils down to performing a few logarithmic operations. The OMT is also a flexible data structure, as values and keys can have process-specific structures. Leaves (L_i) of an OMT form a complete collection. Any query can be answered by providing a leaf that exists in the tree. Furthermore, OMTs can be hierarchical. The value (v_i) (Figure 3.8) of a leaf in a collection can be the root of a nested OMT. Any number of such nested levels can be used. TMMC can have both dynamic and static OMTs. Dynamic OMT stores data items of the process and static OMT storesfixed state-transition rules.

Using OMT, the prover can store an unlimited number of data items (as leaves) in an OMT. They can offer succinct proofs of existence or non-existence of key value of any key and highest/lowest keys in any (possibly nested) collection. Some of the notations used by TMMC are as follows:

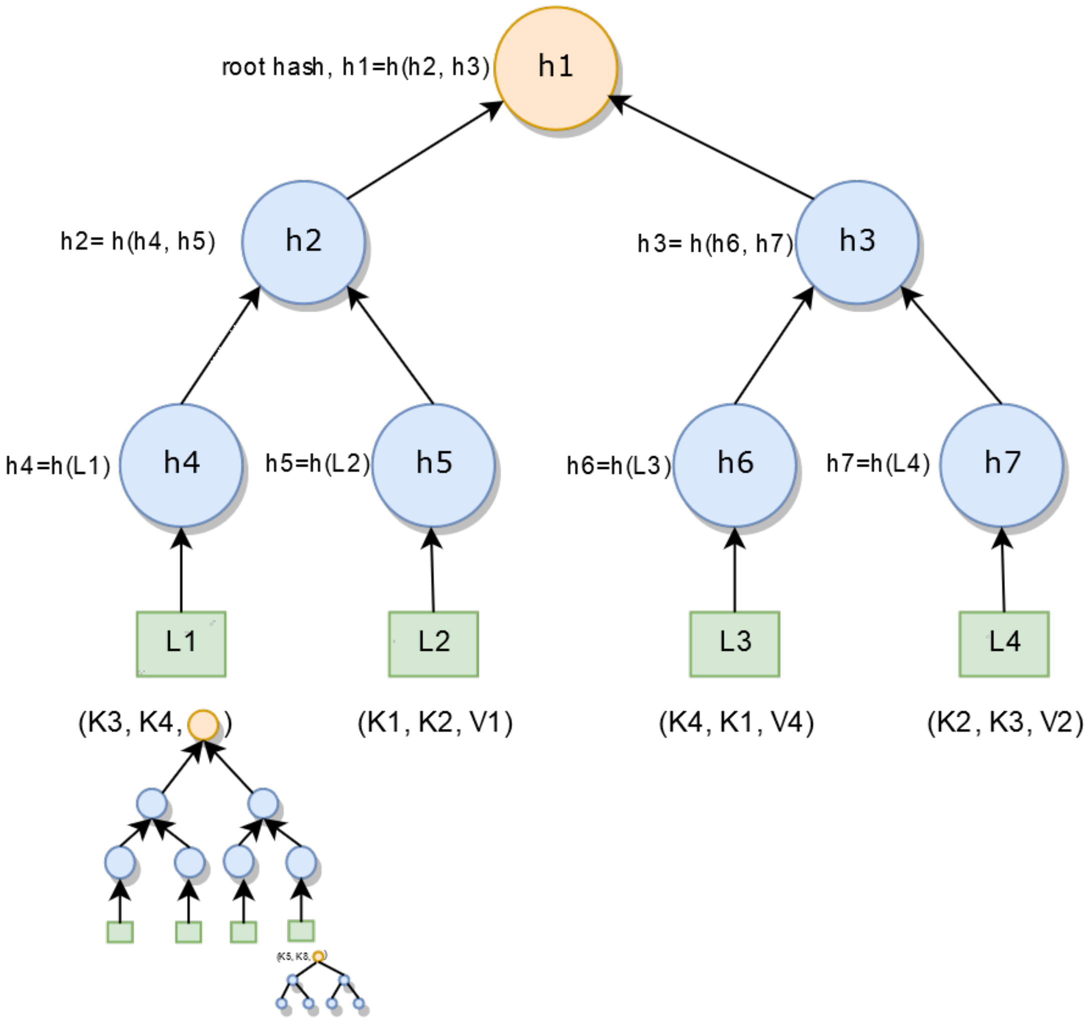


Figure 3.8

OMT hierarchy

- $(k, v)_{k_n} \in r$ or $(k, v) \in r$: key-value pair (k, v) exists in an OMT collection with root r
- $k \notin r$: Key k does not exist in an OMT collection with root r

The overhead for memory read/write/update/delete in TMMC is $\mathcal{O}(\log_2 N)$ (N = number of leaves in the OMT) as opposed to the constant time in the von Neumann model. Both the existence and non-existence of a leaf in OMT can be checked in logarithmic time. Finding the highest/lowest/next keys of the OMT also takes logarithmic time because of the inherent structure of OMT. Storing OMT takes up the same amount of space as any data structure. Therefore, the main differences lie in accessing items of the OMTas it involves checking the correctness of each read/write.

3.4 State-transition Functions

In the proposed model, state-transition functions are considered atomic. The atomic nature of these state-transition rules allow the verifiers to consider one state-transition rule at a time to ensure its integrity. This nature also allows verifiers who only have limited memory and processing unit to execute and verify any specific state change at any time. Individual verification of each atomic state-transition function (ASF) yields the overall correctness of the system.

Each ASF results in an update to a small number of leaves in OMT, and consequently, the root of the tree. In TMMC, the state-change functions are deliberately constrained to be simple, to enable provers to offer concise proofs of correctness of any state-change, to verifiers.

Any atomic state-change function (ASF) can be characterized by

1. Input or the trigger for execution of the state-change;
2. Preconditions to be satisfied (before the state-change); and

3. Post-conditions following the state-change.

In TMMC, the state of a process is captured by an OMT with root s. For TMMC state-change functions (that modify the root s), the trigger is a transaction of a specific type. Preconditions are a set of readily verifiable existence/ non-existence /highest / lowest rules regarding specific items in the OMT collection with root s. Post-conditions dictate how some of the pre-conditions will have to be modified to update the OMT root to s' (to satisfy post-conditions).

As an example, consider a state-change function defined as follows:

1. Transaction $T : m \parallel A \parallel B \parallel 5$
2. Preconditions: $(A, v_a > 5) \in s$, $(B, v_b) \in s$
3. Post-conditions: $v_a - = 5$, $v_b + = 5$

Such a state-change function could represent a transaction of type m that requests a transfer of 5 units from an account A to an account B. The preconditions are that key-value pairs $(A, v_a > 5)$ and (B, v_b) should exist in the OMT with root s. The post-conditions demand that the root be updated to reflect the updates to the values of 2 key-value pairs.

3.4.1 Notations

We represent an atomic state-transition function as a set of pre-conditions and post-conditions. We use standard notations to represent the pre-conditions and post-conditions. These notations can be used as formal specification language to write pre-conditions and post-conditions for state-transition rules of any process. In this section, we outline the standard notation used for writing pre-conditions and post-conditions.

- **Existence** $\in$: To denote a certain key k exists in an OMT tree with root r, we write $k \in r$ which reads k exists in an OMT tree with root r. We can also take the corresponding value into account while checking for the existence i.e. $(k, v) \in r$ which reads key k with value v exists in an OMT tree with root r.

- **Non-existence** $\notin$: To denote a certain key k does not exist in an OMT tree with root r, we write $k \notin r$ which reads k does not exist in an OMT tree with root r.

- **Inserting data item**: To insert a new *data item* k with value v to the OMT, first it needs to be made sure that the data item does not exist in the OMT with root r i.e. $k \notin r$. Then we add a placeholder with value 0 for the newly added item in the OMT i.e. $(k, 0) \in r$. After that, we update the value from 0 to v.

- **Deleting data item**: To delete a data item k having value v from the OMT, first it needs to be made sure that the data item exists in the OMT with root r i.e. $k \in r$. Then we update its value to 0 i.e. $(k, 0) \in r$.

- **Updating data item**: To update the value v of a *data item* k to value v' from the OMT, first it needs to be made sure that the *data item* exists in the OMT with root r i.e. $(k, v) \in r$. Then we update its value to v' i.e. $(k, v') \in r$.

- **Increasing data item by 1**: To increase the value v of a data item k to $v + 1$ in the OMT, first it needs to be made sure that the data item exists in the OMT with root r i.e. $(k, v) \in r$. Then we update its value to $v + 1$ i.e. $(k, v + 1) \in r$.

- **Decreasing data item by 1**: To decrease the value v of a data item k to $v-1$ in the OMT, first it needs to be made sure that the data item exists in the OMT with root r i.e. $(k, v) \in r$. Then we update its value to $v-1$ i.e. $(k, v-1) \in r$.

- **Use of registers**: In general, the small constant memory size of verifiers can be seen as a small fixed number of registers. Registers are denoted using capital letters, e.g. R_0, R_1, R_2 etc. Some registers may be OMT roots.

3.4.2 Building Blocks of State-transition Functions

1. **Inputs**: Inputs are unconstrained data items (UDI) which trigger the state-transition function. Example: x wants to send m dollars to y. The input of the state-transition function should be as follows:

$$[update; x; y; m]_w; update = \text{operation code}; w = \text{signature of the authorized entity}$$

2. **Pre-conditions**: Pre-condition is the state system that needs to be in before the state-transition. Example: Let, r is the root-hash of an OMT tree representing its current state. Data item x and y should both exist in OMT with root r i.e $(x \in r, y \in r)$ and the account balance b_x of x should be greater or equal to m. Therefore the pre-conditions can be written as follows:

$$(x, b_x)_{x_n} \in r, b_x > m; x_n = \text{next account key of } x$$

$$(y, b_y)_{y_n} \in r; y_n = \text{next account key of } y$$

3. **Post-conditions**: Post-conditions are the state of the system after the successful execution of the state-transition function. The root of the OMT changes after these post-conditions is

executed. Example: Updated balance (b_{x_0} and b_{y_0}) in x and y's account after the balance transfer.

$$(x, b_{x0})_{x_n} \in r$$

$$(y, b_{y0})_{y_n} \in r$$

3.4.3 Utility Functions of TMMC

The state-transition rules make well-defined changes to the state of the system. However, to make the state-transition function more readable and compact we express most state transition functions using a small set of simple utility functions. These functions, when considered separately, are also atomic. A brief list of some of these utility functions is given in this section.

- **get_min(R):** This utility function returns the minimum key from the OMT with root hash R. The inherent structure of OMT allows finding the minimum key in $\mathcal{O}(\log_2 N)$ time (N =number of keys). Below is a notion of getting the minimum key in an OMT and storing it in register MIN.

  Input: MIN

  Pre: $MIN = get\_min(R), MIN \in R$

  Post: $MIN \in R$

- **extract_min(R):** This utility function extracts the minimum key from the OMT with root hash R. After the extraction, the key will no longer exist in OMT R. Non-existence is denoted by setting the corresponding value to zero. The inherent structure of OMT allows extracting the minimum key in $\mathcal{O}(\log_2 N)$ time (N =number of keys).

Input: MIN

Pre: $MIN = extract\_min(R), MIN \in R$

Post: $MIN \notin R$ or $(MIN, 0) \in R$

- **get_max(R):** Similar to the get_min function, this utility function gets the maximum key from the OMT with root hash R. The inherent structure of OMT allows finding the maximum key in $\mathcal{O}(\log_2 N)$ time (N =number of keys). Below is a notion of getting the maximum key in an OMT and storing it in register MAX.

  Input: MAX

  Pre: $MAX = get\_max(R), MAX \in R$

  Post: $MAX \in R$

- **extract_max(R):** Similar to the $extract\_min$ function, this utility function extracts the maximum key from the OMT with root hash R. After the extraction, the key will no longer exist in OMT R. Non-existence is denoted by setting the corresponding value to zero. The inherent structure of OMT allows extracting the maximum key in $\mathcal{O}(\log_2 N)$ time (N =number of keys).

  Input: MAX

  Pre: $MAX = extract\_max(R), MAX \in R$

  Post: $MAX \notin R$ or $(MAX, 0) \in R$

- **split(s):** This function splits a key or a value of a leaf s into multiple items.

Input: s, s_1, s_2

Pre: $s \neq 0, s_1 = split(s)[0], s_2 = split(s)[1]$

Post: $<$ Do something with the split sub-strings $>$

- **next_key(**(k, R)**):** This function returns the next key of k in OMT R

  Input: k_1, k_2

  Pre: $k_1 \in R, k2 = next\_key(k_1, R)$

  Post: $k_1 \in R$

- **determinant(**$x_1, y_1, x_2, y_2, x_3, y_3$**):** This function returns the determinant of the three given points (x_1, y_1), (x_2, y_2) and (x_3, y_3)

  Input: $x_1, y_1, x_2, y_2, x_3, y_3, D$

  Pre: $D = determinant(x_1, y_1, x_2, y_2, x_3, y_3)$

  Post: $<$ Do something with $D >$

- **prev_key(**k_1, R**):** This function returns the previous key of k_1 in OMT R

  Input: k_1, k_0

  Pre: $k_0 \in R, k_0 = prev\_key(k_1, R)$

  Post: $k_0 \in R$

- **does_not_intersect(**a, b**):** This function returns false if line segment a and b intersect with each other and true if they do not. There should be an OMT (e.g R) which stores the end-points of the segments with identifiers a and b

Input: $a, x_a, y_a, b, x_b, x_b, INTRSCT$

Pre: $INTRSCT = 0, (a, (x_a, y_a)) \in R, (b, (x_b, y_b)) \in R)$

Post: $INTRSCT \leftarrow does\_not\_intersect(a, b)$

- **x.contains(y):** This function returns true if string x contains sub-string y, false otherwise.

  Input: $x, y, CONTAINS$

  Pre: $CONTAINS = 0$

  Post: $CONTAINS \leftarrow x.contains(y)$

- **dist(x_1, y_1, x_2, y_2):** This function returns the euclidean distance between the given points.

  Input: x_1, y_1, x_2, y_2, D

  Pre: $D = 0$

  Post: $D \leftarrow dist(x_1, y_1, x_2, y_2)$

3.5 Verification of Correctness

It is important to note that the purpose of TMMC is ultimately to verify the correctness of algorithm execution. In general, the goal of executing an algorithm $\mathcal{O} = f(I)$ for a given input I, is to generate the corresponding output O. Under the TMMC model, the goal is to merely verify that $\mathcal{O} = f(I)$.

In general, verification of the correctness of any algorithm is at most only as complex as the execution of the algorithm. Therefore, the algorithm for verification of correctness of a solution O can be substantially easier than the algorithm for obtaining O.

Thus, a solution O can be determined using the traditional VN model of computing. For such problems, there is often a metric m_o associated with the solution O. Given a solution O with metric m_o (for an algorithm $f()$ operating on input I) the TMMC algorithm is merely intended to establish that the solution is correct and that the metric is indeed m_o. When such algorithms are executed in the blockchain-TMMC, incentivized users may compete to provide the solution with the best metric, and execute the verification algorithm to prove the correctness of the solution.

3.6 Useful TMMC Data Structures

Executing different algorithms require storing the input or output data items into various data structures. The type of these data structures varies depending on the algorithm we are considering. OMT can be used to simulate these different data structures. In this section, we describe some data structure we have used while describing a range of algorithms using TMMC in the next chapters.

- **Heap:** In TMMC, a heap is represented with two OMTs. For example, we want to represent the edge weights of a graph using a heap. The first OMT will contain the edge names as keys and weight as values and the second OMT will store the weights as keys and the number of edges with that weight as values (Table 3.1)

- **Adjacency list:** We use the adjacency list to represent graphs in TMMC where keys are the nodes and the values are nested OMT. This nested OMT has nodes as keys and edge weight as values (Table 3.2)

- **Disjoint set or Union find**: To represent disjoint sets using OMT we need two OMTs. The first one to keep track of the edges where edge identifiers are used as keys and edge weights

Table 3.1

Minimum heap representation

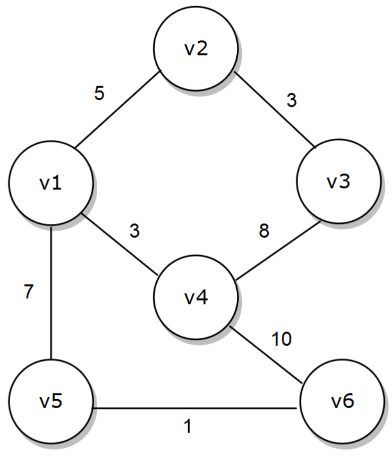
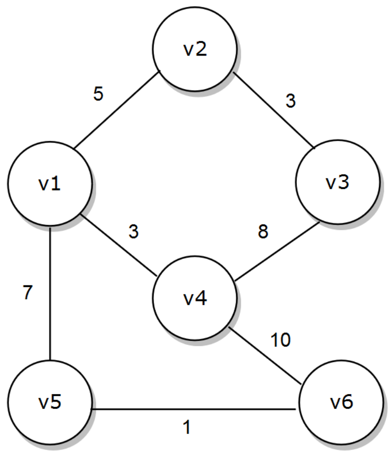
Input Graph	OMT representation
 v2 v1 v3 v4 v5 v6 5 3 3 8 7 10 1	OMT_1 $(v_1v_2, 5)$ $(v_2v_3, 3)$ $(v_3v_4, 8)$ $(v_1v_4, 3)$ $(v_4v_6, 10)$ $(v_5v_6, 1)$ $(v_1v_5, 7)$ OMT_2 $(5, 1)$ $(3, 2)$ $(8, 1)$ $(10, 1)$ $(1, 1)$ $(7, 1)$

Table 3.2

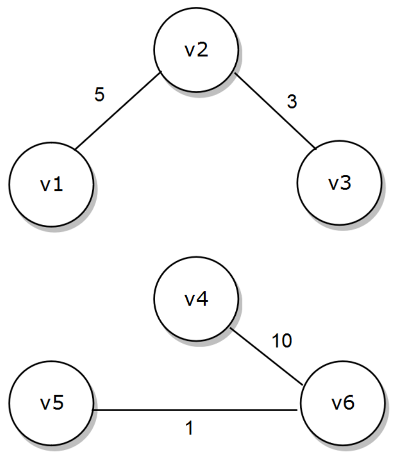
Adjacency list representation

Input Graph	OMT representation
 v2 v1 v3 v4 v5 v6 5 3 3 8 7 10 1	$(v_1, ((v_2, 5), (v_4, 3), (v_5, 7)))$ $(v_2, ((v_1, 5), (v_3, 3)))$ $(v_3, ((v_2, 3), (v_4, 8)))$ $(v_4, ((v_1, 3), (v_3, 8), (v_6, 10)))$ $(v_5, ((v_1, 7), (v_6, 1)))$ $(v_6, ((v_4, 5), (v_5, 3)))$

are used as values. In the second OMT, we keep track of which node belongs to which set. This means nodes are stored as keys and their parent nodes are stored as values. To assign a parent node to a particular node, we first check whether it belongs to an edge where the other endpoint already has a parent. Then we assign that same parent node to this current node. Else, if none of them have an entry in the second OMT, they become the parent of themselves (Table 3.3)

Table 3.3

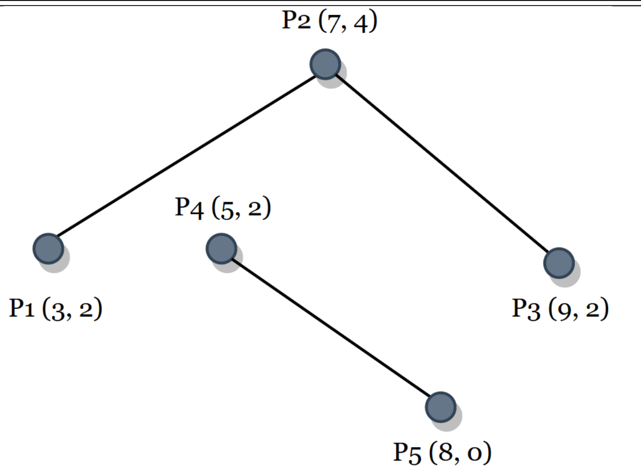
Disjoint set representation

Input Graph	OMT representation
	OMT_1 $(v_1v_2, 5)$ $(v_2v_3, 3)$ $(v_4v_6, 10)$ $(v_5v_6, 1)$ OMT_2 (v_1, v_1) (v_2, v_1) (v_3, v_1) (v_4, v_4) (v_5, v_4) (v_6, v_4)

- **Cartesian points:** To represent cartesian points in OMT, we simply store the point identifier as keys and the coordinates as a value e.g. $x \parallel y$ (Table 3.4)

Table 3.4

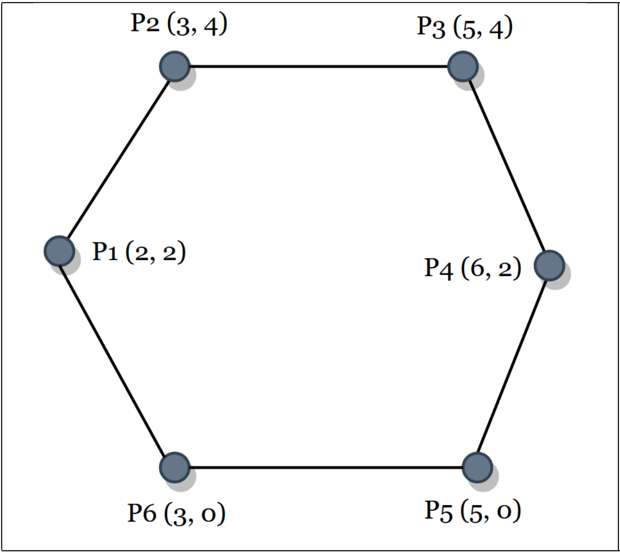
Cartesian points representation

Input Graph	OMT representation
 P2 (7, 4) P4 (5, 2) P1 (3, 2) P3 (9, 2) P5 (8, 0)	$(P_1, 3 \parallel 2)$ $(P_2, 7 \parallel 4)$ $(P_3, 9 \parallel 2)$ $(P_4, 5 \parallel 2)$ $(P_5, 8 \parallel 0)$

- **Polygon:** Polygons are represented by storing their participating points in the OMT, similar to storing cartesian points. However, in the case of points, the points are stored sequentially in a clockwise direction (Table 3.5)

Table 3.5

Polygon representation

Input Graph	OMT representation
 P2 (3, 4) P3 (5, 4) P1 (2, 2) P4 (6, 2) P6 (3, 0) P5 (5, 0)	$(P_1, 2 \parallel 2)_{P_2}$ $(P_2, 3 \parallel 4)_{P_3}$ $(P_3, 5 \parallel 4)_{P_4}$ $(P_4, 6 \parallel 2)_{P_5}$ $(P_5, 5 \parallel 0)_{P_6}$ $(P_6, 3 \parallel 0)_{P_1}$

CHAPTER IV

TMMC WORKFLOW

TMMC model provides a reliable platform to execute information system processes. Complex information systems can be seen as a collection of processes that changes the state of the system, causing the information to change within the system. To illustrate the transformation of conventional process execution to atomic state-transitions functions (ASF) using the TMMC model, we take into account known and widely used algorithms. In this section, we describe in detail, how a conventional algorithm can be converted to a sequence of atomic state-transition functions.

4.1 Algorithm to Atomic State-transition Functions

The first step is to identifying the algorithm to convert it to state-transition functions. Algorithms from any domain of interest can be chosen. In traditional architecture, algorithms are executed by calling the associated procedures. These procedures can have underlying sub-procedures that may not be evident from observing the procedure call stack.

In TMMC, we choose a process that solves a problem or changes a set of data items to compute something through the execution of the process. For example lets take into account a popular algorithm (A) that solves for a convex hull H given a set of N points $(p_1, p_2, \ldots, p_N)$. Given N points, the convex hull represents the subset of N points that forms the smallest convex polygon containing all the N points. The algorithm is given in Figure 6.2.

```
Algorithm 5: Graham's Scan Algorithm ($P$)
  Result: A list of vertices in clockwise order representing $CH(P)$
  Sort the points by $x$-coordinate, resulting in a sequence $p_1$,..., $p_k$
  Insert $p_1$ and $p_2$ in list $L_{upper}$, with $p_1$ as the first point
  for $i \leftarrow 3$ to $k$ do
      Append $p_i$ to $L_{upper}$
      $n = length(L_{upper})$
      while $n > 2$ and the $p_{n-2}, p_{n-1}, p_n \in L_{upper}$ do not make a right turn do
          Delete $p_{n-1}$ from $L_{upper}$
      end
  end
  Insert $p_k$ and $p_{k-1}$ in list $L_{lower}$, with $p_k$ as the first point
  for $i \leftarrow k - 2$ down to 1 do
      Append $p_i$ to $L_{lower}$
      $m = length(L_{lower})$
      while $m > 3$ and $p_{m-2}, p_{m-1}, p_m \in L_{lower}$ do not make a right turn do
          Delete $p_{m-1}$ from $L_{lower}$
      end
  end
  Remove $p_0$ and $p_m$ from $L_{lower}$ to avoid duplication in the the upper and lower hull
  Append $L_{lower}$ to $L_{upper}$, and call the resulting list $L$
  return $L$
```

Figure 4.1

Graham's scan algorithm

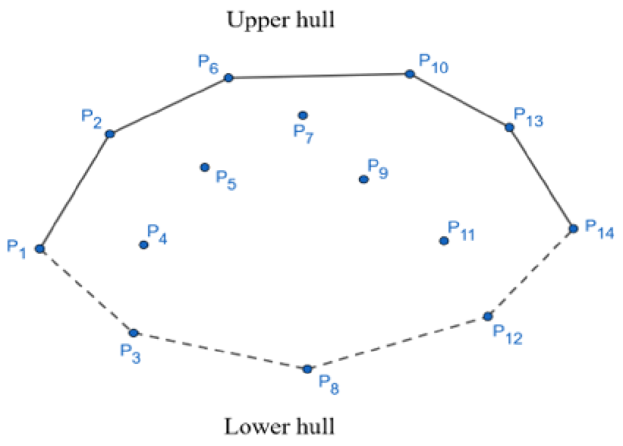


Figure 4.2

Convex hull computation

4.2 Identification of Data Items

The next step is to determine the data items involved in the algorithm. First, there are unconstrained data items (UDI) that are basically the inputs to the algorithm given from outside. These UDIs will work as triggers to invoke the state-transition functions. Correct UDIs will ensure correct execution of the state-transition functions. For example: at the beginning of executing the state-transition functions of the convex hull algorithm, the OMT root r of the input set of points $(p_1, p_2, \ldots, p_N)$ is provided to invoke the first *initialization* function.

Then there are constrained data items (CDI), which are the critical data items that are changed by the successful execution of the permitted state-transition functions. These CDIs are stored as leaves of the ordered Merkle tree and stored by the untrusted prover. For example, computing convex hull requires building a sorted list of x coordinates that are stored as OMT leaves (CDIs). These leaves are modified through the execution of state-transition functions.

4.3 Configuration of Data Structure

In this step, the first thing to decide is the number of OMT required to execute the convex hull algorithms. A fixed number of OMTs is used for a particular algorithm and specific *registers* are set aside for storing the root of these OMTs. These registers can store a very small amount of data and for storing the OMT roots this small space is enough. Moreover, constant registers can also be defined to store temporary values to keep track of the algorithm execution e.g. states, constants, etc.

For convex hull algorithm, three OMTs are needed which are stored in register R_0, R_1 and R_2. In addition, four registers are used namely P_{c-1}, P_c, P_{c+1} and P_{c+2} to store four points that are consecutive in terms of x coordinates.

4.4 Process Representation as State-transition Function

The procedural algorithmic are then converted into atomic state-transition functions. There can only be a fixed number of state-transition functions for executing a particular process. Each of these state-transition function contains rules that are strictly pre-defined and also stored in a static OMT. These state-transitions are atomic which means

- They are independently executable
- They either successfully executed or not executed at all
- They do not contain any rules that are repeated for more than one data items (no looping)
- They can be independently executed as many times as required

These state-transition functions are lightweight and can be readily executed by anyone who does not possess high computational resources. Single state-transition functions have three parts.

1. **Inputs**: UDIs for the state-transition function
2. **Pre-condition**: State the system should be before the execution of the state-transition function
3. **Post-condition**: State the system changes to after the successful execution of the state-transition function

Each state-transition function is associated with a particular complexity that can be computed at this point. Point to note that each of the state-transition function will have a logarithmic component attached to it as read/write in TMMC takes logarithmic time. The algorithm i.e. Graham's scan, that solves the convex hull problem can be represented with four state-transition functions.

- Initialization of input points
- Right/Left Turn detection of 3 adjacent points (when ordered by X-coordinate)for upper convex hull (UCH) computation
- Right/Left Turn detection of 3 adjacent points(when ordered by X-coordinate)for lower convexhull (LCH) computation
- Merging two points (one from UCH and one from LCH)

4.5 Python Implementation of Atomic State-transition Functions

The state-transition functions for a particular process can be readily executed by a trusted hardware or as transactions in a blockchain network to ensure the correct execution of algorithm that computes the convex hull. However, for testing purpose, we can map the state-transition functions to python functions. These python functions have the inputs as function parameters, pre-conditions as *if* condition and post-conditions as statement that are executed once the if conditions are satisfied.

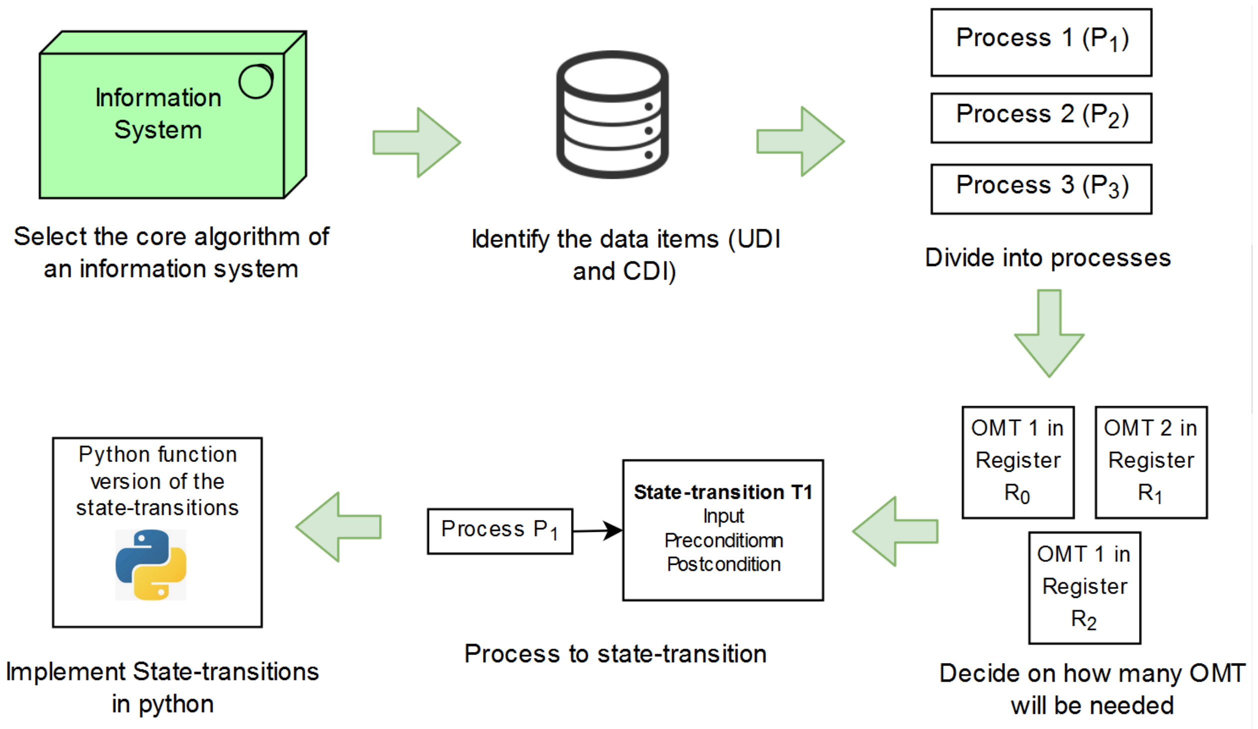

Information System
Select the core algorithm of an information system
Identify the data items (UDI and CDI)
Process 1 (P_1)
Process 2 (P_2)
Process 3 (P_3)
Divide into processes
OMT 1 in Register R_0
OMT 2 in Register R_1
OMT 1 in Register R_2
Decide on how many OMT will be needed
Process P_1
State-transition T1
Input
Preconditiomn
Postcondition
Process to state-transition
Python function version of the state-transitions
Implement State-transitions in python

Figure 4.3

TMMC workflow

CHAPTER V

GRAPH ALGORITHMS

Graph theory is a significant area of computer science. Many problems in the field of computer science require some sort of graph formulation in order to solve the problem. In many large-scale problems such as representing network connections, social networks, road networks, graphs are extensively used. Graphs are the starting point to compute many distance measures as well. One of the most popular usages of graph algorithms is to calculate the shortest path between nodes and computing minimum spanning tree of a given graph. We have focused on these two types of algorithms in this chapter.

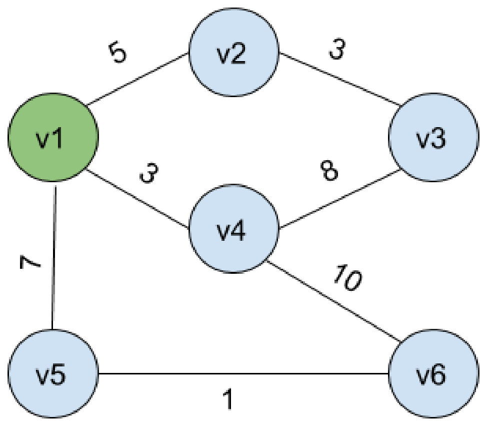


Figure 5.1

A sample weighted graph

Two of the most famous algorithms for solving single-source shortest path problems are Dijkstra's and Bellman-Ford algorithm [14]. Prim's and Kruskal's algorithm [14] are well-known for computing the minimum spanning tree (MST) of an input graph. Computing the solution of these algorithms and verifying the given solution require a similar effort. Hence we convert the original graph algorithms to state-transition functions using the TMMC model. In this section we consider Dijkstra's algorithm [14] a widely used algorithm for computing the minimum spanning tree of a given graph. We convert the processes mentioned in the original Dijkstra's algorithm into state-transition functions using the TMMC model. We also describe the state-transition functions of another popular algorithm i.e. Prim's Algorithm [14] for computing the minimum spanning tree of an input graph.

5.1 Graph as Inputs to TMMC

The way we give the graph as input depends large on the algorithm itself. For example for Dijkstra we expect our input to be a set of nodes (as keys of OMT leaves) along with their weighted neighbor lists (as values of the OMT leaves). On the other hand for Kruskal's, we are mostly interested in the edges over nodes, therefore the OMT leaves contain edge names (two endpoints) as keys and edge weights as values [Table 5.1]. In some cases, the graph edges can be one-directional or bi-directional. In the latter case, we consider edges two times with different weights.

5.2 Dijkstra's Algorithm

Dijkstra's algorithm [14] is used to find the shortest path from a given start node to the destination. This algorithm does not consider any negative edge thus cannot be applied in such cases. Dijkstra's algorithm solves the single-source shortest path as opposed to all-pair shortest path. The

Table 5.1

Two ways to input sample graph in Figure 5.1 to TMMC

Dijkstra	Kruskal
$(v_1$, $((v_2, 5)$, $(v_4, 3)$, $(v_5, 7)))$	$(v_1v_2, 5)$
$(v_2$, $((v_1, 5)$, $(v_3, 3)))$	$(v_2v_3, 3)$
$(v_3$, $((v_2, 3)$, $(v_4, 8)))$	$(v_1v_4, 3)$
$(v_4$, $((v_1, 3)$, $(v_3, 8)$, $(v_6, 10)))$	$(v_3v_4, 8)$
$(v_5$, $((v_1, 7)$, $(v_6, 1)))$	$(v_1v_5, 7)$
$(v_5$, $((v_4, 10)$, $(v_5, 1)))$	$(v_4v_6, 10)$
	$(v_5v_6, 1)$

input for Dijkstra's algorithm is a graph $G = (V, E)$ with a given start node s. Cost to reach every node from the start node is calculated by constantly updating the path cost as we explore all the edges. The algorithm is given in Figure 5.2.

5.2.1 Predicates and State-transition Functions

Execution of Dijkstra's algorithm to compute single-source shortest path require four OMTs. The roots of these OMTs are stored in five registers (R_0, R_1, R_2, R_3, R_4) along with some additional registers to store constant (C, D).

- R_0 stores the root of the OMT containing the input graph with the node as key and the nested OMT representing the connected edges as value.

- R_1 stores the root of nested OMT from R_0 containing the neighboring node as key and the edge weight as value.

- R_2 stores the root of OMT containing the node identifiers as key and the path cost from the start node as value.

```
Algorithm 1: Dijkstra $(G, s, w)$
  Result: Single source shortest path
  $dist[s] \leftarrow 0$
  for all $v \in V[G]$ do
    $dist[v] = \infty$
  end
  $S = \emptyset$
  $Q = V[G]$
  while $Q \neq \emptyset$ do
    $u = extract\_min(Q)$ $S = S \cup \{u\}$
    for all $v \in neighbors[u]$ do
      if $dist[v] > dist[u] + w(u, v)$ then
        $d[v] \leftarrow d[u] + w(u, v)$
      end
    end
  end
```

Figure 5.2

Dijkstra's algorithm

- R_3 stores the root of OMT containing the edge weights as key and the count of edge weights as value. R_2 and R_3 together form the minimum heap data structure.
- R_4 stores the final costs to reach every node from the start node
- C stores the current node being explored
- D stores the cost to reach the current node

We need four state-transition functions to execute Dijkstra using TMMC.

1. **Initialization (F_0):** This function initializes the root r of input graph OMT to R_0. After the execution of F_0, $R_0 = 0$ will no longer satisfy.
2. **Assign start node (F_1):** F_1 removes the given start node n from R_0 and assigns it as a current node. The distance of start node n to itself is set to $D = 1$ (as value 0 has a special

Table 5.2

Predicates of Dijkstra's algorithm

Registers	Python Variables

$R_0 : (n, v)$, n=node, v=nested OMT root
$R_1 : (n', v')$, n'=node, v=weight of edge (n, n'), where $n' = neighbor(n)$
$R_2 : (n, w)$, n=node, w=path cost to reach n
$R_3 : (w, ct_w)$, w=edge weight, ct_w=weight count
$R_4 : (n, w)$, n=node, w=final path cost to reach n
C : Stores the current processing node (n) from R_0
D : Stores the cost to reach current node in C from the starting node
	`R0 = {}`
	`R1 = {}`
	`R2 = {}`
	`R3 = {}`
	`R4 = {}`
	`C = D = 0`

Table 5.3

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0
Pre: $R_0 = R_1 = R_2 = R_3 = R_4 = C = 0$
Post: $R_0 \leftarrow r_0$
	`def init(r0):`
	`    if not R0 and not R1 and not R2`
	`    and not R3 and not R4:`
	`        R0=r0`

interpretation). R_1 stores the value v of n in R_0 representing edges emanating from the current node n.

Table 5.4

Assign start node

F_1: Assign start node	F_1: Python Equivalent

Input: n, v, C, D
Pre: $R_0 \neq 0$, $R_1 = 0$, $(n, v) \in R_0$, $C = D = 0$
Post: $n \notin R_0$, $R_1 \leftarrow v$, $(n, 1) \in R_2$, $(1, 1) \in R_3$, $(n, 1) \in R_4$, $C = n$, $D = 1$
	`def assign_start_node(n, v, C, D):`
	`    if R0!=0 and R1!=0 and R0[n]==v`
	`    and C==0 and D==0:`
	`        R0[n]=0`
	`        R1=v`
	`        R4[n]=1`
	`        C=n`
	`        D=1`

3. **Update node cost (F_2):** F_2 is the function to update the path cost of nodes in R_2. There can be several cases therefore this function is divided case-wise into five functions i.e. F_{2a} to F_{2e}. These functions are invoked multiple times based on the number of nodes and their connected edges.

4. **Extract minimum weighted edge (F_3):** This function extracts the minimum edge connected to the current node C from R_2 and assigns the adjacent node as the new current node.

5.2.2 Complexity Analysis

Dijkstra's algorithm finds the shortest path from the start node to every other node in a graph G with V nodes and E edges. Using conventional architecture, the complexity of solving Dijkstra's

Table 5.5

Update node cost, case (a) (Discard, node exists in output R_5)

F_{2a}: Update node cost, case (a)	F_{2a}: Python Equivalent

Input: n, v
Pre: $(n, v) \in R_1, n \in R_4$
Post: $n \notin R_1$
	`def update_node_cost_a(n, v):`
	`    if R1[n]==v and n in R4:`
	`        R1[n]=0`

Table 5.6

Update node cost, case (b) (Insert, no entry exist in R_2)

F_{2b}: Update node cost, case (b)	F_{2b}: Python Equivalent

Input: n, v
Pre: $(n, v) \in R_1, n \notin R_2$
Post: $n \notin R_1, (n, v + D) \in R_2, (v + D, 1) \in R_3$
	`def update_node_b(n, v):`
	`    if R1[n]=v and R2[n]==0:`
	`        R[n]=0`
	`        R2[n]=v+D`
	`        R3[v+D]=1`

Table 5.7

Update node cost, case (c) (Discard, lower cost exists in R_2)

F_{2c}: Update node cost, case (c)	F_{2c}: Python Equivalent

Input: n, v, w, w_{old}
Pre: $(n, v) \in R_1$, $(n, w_{old}) \in R_2$, $v + D = w \geq w_{old}$
Post: $n \notin R_1$
	`def update_node_c(n, v, w_old):`
	`    if R1[n]==v and R2[n]==w_old`
	`   and v+D>=w_old:`
	`        R1[n]=0`

Table 5.8

Update node cost, case (d) (Insert, new cost is lower and does not exist in R_3)

F_{2d}: Update node cost, case (d)	F_{2d}: Python Equivalent

Input: $n, v, w, w_{old}, ct_{old}$
Pre: $(n, v) \in R_1$, $(n, w_{old}) \in R_2$, $(w_{old}, ct_{old}) \in R_3$, $v + D = w < w_{old}$, $w \notin R_3$
Post: $n \notin R_1$, $(n, w) \in R_2$, $(w_{old}, ct_{old} - 1) \in R_3$, $(w, 1) \in R_3$
	`def update_node_cost_d(n, v, w_old,`
	`     ct_old, D):`
	`    if R1[n]==v and R2[n]==w_old`
	`   and R3[w_old]==ct_old and v+D<`
	`   w_old and w not in R3:`
	`        R1[n]=0`
	`        R2[n]=w`
	`        R3[w_old]=ct_old-1`
	`        R3[w]=1`

Table 5.9

Update node cost, case (e) (Update, new cost is lower and exists in R_3)

F_{2e}: Update node cost, case (e)	F_{2e}: Python Equivalent

Input: $n, v, w_{old}, ct_{old}, ct_{new}$
Pre: $(n, v) \in R_10$, $(n, w_{old}) \in R_2$, $(w_{old}, ct_{old}) \in R_3$, $v + D = w < w_{old}$, $(w, ct_{new}) \in R_3$
Post: $n \notin R_1$, $(n, w) \in R_2$, $(w_{old}, ct_{old} - 1) \in R_3$, $(w, ct_{new} + 1) \in R_3$
	`def update_node_cost_e(n, v, w_old,`
	`     ct_old, ct_new, D):`
	`    if R1[n]=v and R2[n]=w_old and`
	`   R3[w_old]=ct_old and v+D<w_old`
	`   and R3[w]=ct_new:`
	`        R1[n]=0`
	`        R2[n]=w`
	`        R3[w_old]=ct_old-1`
	`        R3[w]=ct_new+1`

Table 5.10

Extract minimum weighted edge

F_3: Extract minimum weighted edge	F_3: Python Equivalent

Input: n, v, w, ct, C, D
Pre: $R_0 \neq 0$, $(n, v) \in R_0$, $R_1 = 0$, $C \neq 0$, $C \in R_0$, $(n, w) \in R_2$, $(w, ct) \in R_3$, $w = get\_min(R_3)$, $n \notin R_4$
Post: $n \notin R_0$, $C = n$, $R_1 \leftarrow v$, $D \leftarrow w$, $n \notin R_2$, $(w, ct-1) \in R_3$, $(n, w) \in R_4$
	`def extract_min_edge(n, v, w, ct, C, D):`
	`    if not is_reg_empty(R0) and R0[n]=v and is_reg_empty(R1) and C !=0 and C in R0 and R2[n]=w and R3[w]=ct and w=get_min(R3) and n not in R4:`
	`        R0[n]=0`
	`        C=n`
	`        R1=v`
	`        D=w`
	`        R2[n]=0`
	`        R3[w]=ct-1`
	`        R4[n]=w`

algorithm is $\mathcal{O}(V^2)$; $\mathcal{O}(E \log_2 V)$ if we use adjacency list representation of the graph. In our representation, each atomic function is associated with different complexity.

The initialization function F_0 is invoked only on time in $\mathcal{O}(1)$ time. F_1 removes the given start node n from R_0 and assigns it as a current node. The distance of start node n to itself is set to D = 1 (as value 0 has a special interpretation). R_1 stores the value v of n in R_0 representing edges emanating from the current node n. It is also invoked one time ($\mathcal{O}(1)$).

F_2 is the function to update the path cost of nodes in R_2. There can be several cases therefore this function is divided case-wise into five functions i.e. F_{2a} to F_{2e}. These functions are invoked multiple times based on the number of nodes and their connected edges. The complexity of this function is $\mathcal{O}(VE \log_2 E)$ as we read through all the edges that are connected to each node. Moreover, in TMMC, every edge read operation takes $\mathcal{O}(\log_2 E)$ as opposed to the conventional constant time reading.

F_3 extracts the minimum edge connected to the current node C from R_2 and assigns the adjacent node as the new current node. In TMMC, extracting maximum and minimum from the OMT takes logarithmic time. So here this function extract minimum edge for every node ($\mathcal{O}(V \log_2 E)$.

5.3 Bellman Ford's Algorithm

Bellman-Ford algorithm [14] solves single-source shortest path problems in a weighted directed graph. The end results of the Bellman-Ford is the same as that of Dijkstra meaning they both give the shortest path to all vertices from the start vertex. However, Bellman-Ford algorithm is able to handle graphs with negative edges and returns no solution when there is a negative cycle as negative cycle can continue to reduce the path cost forever. Bellman-Ford is not a greedy algorithm unlike Dijkstra.

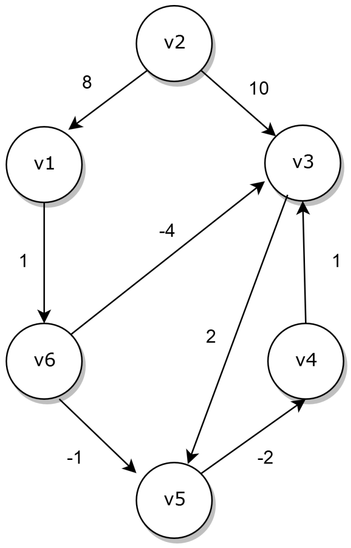


Figure 5.3

A sample directed graph with negative edges

Lets take $G = (V, E)$ as an input graph. G has n vertices. The goal is to calculate the shortest distance d_{sd} where $d_{sd} = dist(V_s, V_d); V_s, V_d \in V$. The algorithm iterates $|V| - 1$ time as there can be at most $|V| - 1$ edges in the path from the V_s to V_d. Any number of edges more than that would mean there is a cycle (repeated vertex). There is no solution to the algorithm if it is a negative cycle. In the case of a positive cycle, it can be removed to reduce the total cost from Vs to V_d. Bellman-Ford's algorithm takes a graph G as input and results in the shortest paths to reach every node from the starting node. Figure 5.3 gives a graph G with nodes $(v_1, \ldots, v_6)$ where v_1 is the starting node. The number on the edges between two nodes denotes the cost to reach to node v_j from v_i where $(i \neq j)$. G is a simple directed graph with no cycles.

Algorithm 2: Bellman Ford (G, s, w) with no negative cycles

```
Result: Single source shortest path
for all $v \in V[G]$ do
    $dist[v] = \infty$
    $prev[v] = Null$
end
$dist[s] = 0$
repeat $|V| - 1$ times for all $e \in E[G]$ do
    update(e)
end
update($(u, v) \in E[G]$){
    $dist(v) = min dist(v), dist(u) + w(u, v)$
}
```

Figure 5.4

Bellman-Ford's algorithm

5.3.1 Predicates and State-transition Functions

Execution of Bellman-Ford's algorithm to compute single-source shortest path require four OMTs. The roots of these OMTs are stored in five registers (R_0, R_1, R_2, R_3, R_4) along with some additional registers to store constant (C, D).

- R_0 stores the root of the OMT containing the input graph with the node as key and the nested OMT representing the connected edges as value.
- R_1 stores the root of nested OMT from R_0 containing the neighboring node as key and the edge weight as value.
- R_2 stores the root of OMT containing the node identifiers as key and the path cost from the start node as value.
- R_3 stores the root of OMT containing the edge weights as key and the count of edge weights as value. R_2 and R_3 together form the minimum heap data structure.
- C stores the current node being explored
- D stores the cost to reach the current node
- $COUNT$ keeps track of the $|V - 1|$ times iteration

We need four state-transition functions to execute Bellman-Ford's using TMMC.

1. **Initialization (F_0):** This function initializes the root r_0 of input graph OMT to R_0. After the execution of F_0, $R_0 = 0$ will no longer satisfy. It assign the current/start node C to the lowest key of R_0. The distance of start node C to itself is set to $D = 1$ (as value 0 has a

Table 5.11

Predicates of Bellman-Ford's algorithm

Registers	Python Variables

$R_0 : (n, v)$, n=node, v=nested OMT root
$R_1 : (n', v')$, n'=node, v=weight of edge(n, n'), where $n' = neighbor(n)$
$R_2 : (n, w)$, n=node, w=path cost to reach n
$R_3 : (w, ct_w)$, w=edge weight, ct_w=weight count
C : Stores the current processing node (n) from R_0
D : Stores the cost to reach current node in C from the starting node
$COUNT$: Keeps track of the $\|V-1\|$ times iteration
	`R0 = {}`
	`R1 = {}`
	`R2 = {}`
	`R3 = {}`
	`C = D = COUNT = 0`

special interpretation). R_1 stores the value v of C in R_0 representing edges emanating from the current node C.

2. **Update node cost (F_1):** F_1 is the function to update the path cost of nodes in R_2. There can be several cases therefore this function is divided case-wise into four functions i.e. F_{1a} to F_{1d}. These functions are invoked multiple times based on the number of nodes and their connected edges.

3. **Choose next node (F_2):** This function chooses the next node to explore edge in R_0 and assigns its edge list to R_1. It only chooses the next node when all the edges of the previous C have been considered to update the edge cost in R_2. This function can also have two cases. Once we finish considering all the edges of all the nodes in R_0, we again reiterate the process $|V - 1|$ time to get the final path cost in R_2.

Table 5.12

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0, v, C, D
Pre: $R_0 = R_1 = R_2 = R_3 = 0$
Post: $R_0 \leftarrow r_0$, $C = get\_min(R_0)$, $(C, v) \in R_0$, $D = 1$, $R_1 \leftarrow v$, $(C, D) \in R_2$, $(D, 1) \in R_3$
	`def init(r0, v, C, D):`
	`    if not R0 and not R1 and not R2`
	`    and not R3:`
	`        R0=r0`
	`        C=get_min(R0)`
	`        D=1`
	`        if v==R0[C]:`
	`            R1=v`
	`            R2[C]=D`
	`            R3[D]=1`

Table 5.13

Update node cost, case (a) (Insert, no entry exist in R_2)

F_{1a}: Update node cost, case (a)	F_{1a}: Python Equivalent

Input: n, v
Pre: $(n, v) \in R_1, n \notin R_2$
Post: $n \notin R_1, (n, v + D) \in R_2, (v + D, 1) \in R_3$
	`def update_node_cost_a(n, v):`
	`    if R1[n]=v and R2[n]==0:`
	`        R[n]=0`
	`        R2[n]=v+D`
	`        R3[v+D]=1`

Table 5.14

Update node cost, case (b) (Discard, lower cost exists in R_2)

F_{1b}: Update node cost, case (b)	F_{1b}: Python Equivalent

Input: n, v, w, w_{old}
Pre: $(n, v) \in R_1$, $(n, w_{old}) \in R_2$, $v + D = w \geq w_{old}$
Post: $n \notin R_1$
	`def update_node_cost_b(n, v, w_old)`
	`    :`
	`    if R1[n]==v and R2[n]==w_old`
	`    and v+D>=w_old:`
	`        R1[n]=0`

Table 5.15

Update node cost, case (c) (Insert, new cost is lower and does not exist in R_3)

F_{1c}: Update node cost, case (c)	F_{1c}: Python Equivalent

Input: $n, v, w, w_{old}, ct_{old}$
Pre: $(n, v) \in R_1$, $(n, w_{old}) \in R_2$, $(w_{old}, ct_{old}) \in R_3$, $v + D = w < w_{old}$, $w \notin R_3$
Post: $n \notin R_1$, $(n, w) \in R_2$, $(w_{old}, ct_{old} - 1) \in R_3$, $(w, 1) \in R_3$
	`def update_node_cost_c(n, v, w_old,`
	`ct_old, D):`
	`if R1[n]==v and R2[n]==w_old`
	`and R3[w_old]==ct_old and v+D<`
	`w_old and w not in R3:`
	`R1[n]=0`
	`R2[n]=w`
	`R3[w_old]=ct_old-1`
	`R3[w]=1`

Table 5.16

Update node cost, case (d) (Update, new cost is lower and exists in R_3)

F_{1d}: Update node cost, case (d)	F_{1d}: Python Equivalent

Input: $n, v, w, w_{old}, ct_{old}, ct_{new}$
Pre: $(n, v) \in R_1$, $(n, w_{old}) \in R_2$, $(w_{old}, ct_{old}) \in R_3$, $v + D = w < w_{old}$, $(w, ct_{new}) \in R_3$
Post: $n \notin R_1$, $(n, w) \in R_2$, $(w_{old}, ct_{old} - 1) \in R_3$, $(w, ct_{new} + 1) \in R_3$
	`def update_node_cost_d(n, v, w_old,`
	`ct_old, ct_new, D):`
	`if R1[n]=v and R2[n]=w_old and`
	`R3[w_old]=ct_old and v+D<w_old`
	`and R3[w]=ct_new:`
	`R1[n]=0`
	`R2[n]=w`
	`R3[w_old]=ct_old-1`
	`R3[w]=ct_new+1`

Table 5.17

Choose next node, case (a)

F_{2a}: Choose next node, case (a)	F_{2a}: Python Equivalent

Input: v, w, C, D
Pre: $next\_key(C, R_0) > C, R_1 = 0, (C, w) \in R_2$
Post: $C = next\_key(C, R_0)$, $(C, v) \in R_0$, $R_1 \leftarrow v$, $D \leftarrow w$
	`def choose_next_node_a(v, w, C, D):`
	`if next_key(C, R0) > C and not`
	`R1 and R2[C]==w:`
	`C=next_key(C, R0)`
	`R0[C]=v`
	`R1=v`
	`D=w`

Table 5.18

Choose next node, case (b)

F_{2b}: Choose next node, case (b)	F_{2b}: Python Equivalent

Input: v, w, ct, C, D
Pre: $next\_key(C, R_0) < C$, $R_1 = 0$, $(C, w) \in R_2$, $COUNT! = length(R_0)$
Post: $C = get\_min(R_0)$, $(C, v) \in R_0$, $R_1 \leftarrow v$, $D \leftarrow w$, $COUNT++$
	`def choose_next_node_b(v, w, C, D):`
	`    if next_key(C, R0) < C and not`
	`   R1 and R2[C]==w and COUNT!=`
	`   length(R0):`
	`        C=get_min(R0)`
	`        R0[C]=v`
	`        R1=v`
	`        D=w`
	`        COUNT=COUNT+1`

5.3.2 Complexity Analysis

Bellman-Ford computes a single-source shortest path for an input graph G with V nodes and E edges. It takes $\mathcal{O}(VE)$ time to execute Bellman-Ford using a conventional model of computing. When using TMMC, every read and write takes $\mathcal{O}(\log_2 N)$ time where N can be either a number of edges or nodes in the graph.

The initialization function F_0 is invoked only on time in $\mathcal{O}(1)$ time. It also performs the $get\_min()$ function in $\mathcal{O}(\log_2 V)$ time.

F_1 is the function to update the path cost of nodes in R_2. There can be four cases i.e. F_{1a} to F_{1d}. These functions are invoked multiple times based on the number of nodes and their connected edges. The function runs VE times and each edge read takes $\mathcal{O}(\log_2 E)$ time ($\mathcal{O}(VE \log_2 E)$ in total).

F_2 choose the next node to explore in R_0 and assigns it to C. This function is repeated V times therefore takes $\mathcal{O}(V \log_2 V)$ time.

5.4 Kruskal's Algorithm

Kruskal's is a greedy algorithm [14] that finds the Minimum Spanning Tree (MST) of a given graph $G = (V, E)$. In Kruskal's algorithm nodes can be considered trees belonging to a forest. The algorithm chooses the edge with the lowest weight repeatedly and connects two different trees of the forest. Thus forests are unified having a particular tree to represent it. The forests are represented as a disjoint-set data structure and they grow by the means of union-find. Let (u, v) be the lowest weighted edge of graph G. u and v belong to forest f_1 and f_2 respectively. As there is an edge exists between them, forest f_1 and f_2 are unified where u is there representative. The complexity is $\mathcal{O}(E \log_2 E)$ (using merge sort to sort the edges). The algorithm is given in Figure 5.5.

```
Algorithm 3: Kruskal $(G, w)$
  Result: Minimum Spanning Tree
  $A \leftarrow \emptyset$
  for each node $v \in V[G]$ do
      MAKE-SET($v$)
  end
  sort the edges of E into non-decreasing order by weight $w$
  $Q \leftarrow V[G]$
  for each edge $(u, v) \in E$, taken in non-decreasing order by weight $w$ do
      if FIND-SET($u$) $\neq$ FIND-SET($v$) then
          $A \leftarrow A \cup \{(u, v)\}$
          UNION($u, v$)
      end
  end
```

Figure 5.5

Kruskal's algorithm

5.4.1 Predicates and State-transition Functions

Execution of Kruskal's algorithm to compute the Minimum Spanning Tree (MST) requires five OMTs by the prover. The roots of these OMTs are stored in five registers (R_0, R_1, R_2, R_3, R_4) by the verifier along with some additional registers to store constant ($C, STATE$).

- R_0 stores the root of the OMT containing the input graph with the node as key and the nested OMT representing the connected edges as value.
- R_1 stores the root of nested OMT from R_0 containing the neighboring node as key and the edge weight as value.
- R_2 stores the root of OMT containing the concatenated node identifiers (that form an edge) as key and the edge weight as value.
- R_3 stores the root of OMT containing the edge weights as key and the count of edge weights as value. R_2 and R_3 together form the minimum heap data structure.
- R_4 stores the root of OMT containing the node(tree) as key and the forest it belongs to as value.
- R_5 stores the root of OMT containing the node(tree) as key and the count of other nodes(trees) that belong to the forest this node represents as value. R_4 and R_5 together form the disjoint set data structure.
- R_6 is the output register storing the edges of the resulting minimum spanning tree

We need six state-transition functions to execute Kruskal's algorithm using TMMC.

Table 5.19

Predicates of Kruskal's algorithm

Registers	Python Variables

R_0 : (n, v), n=node, v=nested OMT root
R_1 : (n', v'), n'=node, v=weight of edge(n, n'), where $n' = neighbor(n)$
R_2 : $(n \parallel n', w)$, $n \parallel n'$=edge, w=edge weight
R_3 : (w, ct_w), w=edge weight, ct_w=weight count
R_4 : (n, n'), n=node, n'=node representing a forest n belongs to
R_5 : (n, ct_n), n=node representing a forest, ct_n=nodes (trees) belonging to that forest
R_6 : $(n \parallel n', f)$, here $n \parallel n'$ forms an edge that contributes to the resulting minimum spanning tree and f is the node identifier for the forest $n \parallel n'$ belongs to
C : Stores the current processing node (n) from R_0
$STATE$: Stores the processing state (integer)
	`R0={}`
	`R1={}`
	`R2={}`
	`R3={}`
	`R4={}`
	`R5={}`
	`R6={}`
	`C=0`
	`STATE=0`

1. **Initialization (F_0):** F_0 initializes the root r of input graph OMT to R_0. After the execution of F_0, $R_0 = 0$ will no longer satisfy. F_0 is invoked only one time.

2. **Disjoint set initialization (F_1):** F_1 initializes each node to a tree (representing a forest) which is initially itself i.e. (n, n) in R_4 and updates R_5 accordingly.

3. **Access edge list (F_2):** F_{2a} removes the given start node n from R_0 and assigns it as a current node. R_1 stores the value v of n in R_0 representing edges emanating from the current node n. F_{2b} does the same steps as F_{2a} but for every other node except the first one.

4. **Create sorted edge list (F_3):** F_3 builds up the minimum heap in R_2 and R_3 using R_0 and R_1. It is called until R_0 and R_1 become empty.

Table 5.20

Initialization

F_0: Initialization	F_0: Python Equivalent
Input: r_0 Pre: $R_0 = R_1 = R_2 = R_3 = R_4 = R_5 = R_6 = C = STATE = 0$ Post: $R_0 \leftarrow r_0$,	`def init(r0):` `if not R0 and not R1 and not R2 and not R3 and not R4 and not R5 and not R6 and not C and not STATE:` `R0=r0`

Table 5.21

Disjoint set initialization

F_1: Disjoint set initialization	F_1: Python Equivalent
Input: n Pre: $n \in R_0$, $n \notin R_3$, $n \notin R_4$, $STATE \neq length(R_0)$ Post: $(n,n) \in R_3$, $(n,1) \in R_4$, $STATE++$	`def disjoint_set_init(n):` `if R0[n]!=0 and R3[n]==0 and R4[n]==0 and STATE != length(R0):` `R3[n]=n` `R4[n]=1` `STATE=STATE+1`

Table 5.22

Access edge list, case (a) first node

F_{2a}: Access edge list, case (a)	F_{2a}: Python Equivalent
Input: n, v, C Pre: $(n,v) \in R_0$, $R_1 = 0$, $C = 0$, $STATE = length(R_0)+1$ Post: $n \notin R_0, R_1 \leftarrow v, C = n$,	`def access_edge_list_a(n, v, C):` `if R0[n]==v and is_reg_empty(R1) and C==0 and STATE=length(R0)+1:` `R0[n]=0` `R1=v` `C=n`

Table 5.23

Access edge list, case (b) other nodes

F_{2b}: Access edge list, case (b)	F_{2b}: Python Equivalent

Input: n, v, C
Pre: $(n, v) \in R_0, R_1 = 0, C \neq 0$
Post: $n \notin R_0, R_1 \leftarrow v, C = n,$
	`def access_edge_list_b(n, v, C):`
	`    if R0[n]==v and is_reg_empty(R1`
	`   ) and C!=0:`
	`        R0[n]=0`
	`        R1=v`
	`        C=n`

Table 5.24

Create sorted edge list, case (a), Weight does not exist in R_3

F_{3a}: Create sorted edge list, case (a)	F_{3a}: Python Equivalent

Input: n, n', wt
Pre: $R_1 \neq 0, (n', wt) \in R_1, n \parallel n' \notin R_2, wt \notin R_3,$ $C = n$
Post: $n' \notin R_1, (n \parallel n', wt) \in R_2, (wt, 1) \in R_3$
	`def create_sorted_edge_list_a(n0,`
	`   n1, wt):`
	`    if not is_reg_empty(R1) and R1[`
	`   n1]==wt and n0_n1 not in R2, wt`
	`   not in R3 and C==n0:`
	`        R1[n1]=0`
	`        R2[n0_n1]=wt`
	`        R3[wt]=1`

Table 5.25

Create sorted edge list, case (b), Weight already exists in R_3

F_{3b}: Create sorted edge list, case (b)	F_{3b}: Python Equivalent

Input: n, n', wt, ct
Pre: $R_1 \neq 0$, $(n', wt) \in R_1$, $n \parallel n' \notin R_2$, $(wt, ct) \in R_3, C = n$
Post: $n' \notin R_1$, $(n \parallel n', wt) \in R_2$, $(wt, ct++) \in R_3$
	`def create_sorted_edge_list_b(n0,`
	`   n1, wt, ct):`
	`    if not is_reg_empty(R1) and R1[`
	`   n1]==wt and n0_n1 not in R2 and`
	`   R3[wt]==ct and C==n0:`
	`        R1[n1]=0`
	`        R2[n0_n1]=wt`
	`        R3[wt]=ct+1`

5. **Extract minimum weighted edge (F_4):** F_4 extracts the minimum edge connected to current node C from R_2.

Table 5.26

Extract minimum weighted edge

F_4: Extract minimum weighted edge	F_4: Python Equivalent

Input: n, n', wt, ct, C
Pre: $R_0 = 0$, $R_2 \neq 0$, $(n \parallel n', wt) \in R_2$, $(wt, ct) \in R_3$, $wt = get\_min(R_3)$, $C \neq 0$
Post: $n \parallel n' \notin R_2, (wt, ct--) \in R_3, C = n \parallel n'$
	`def extract_min(n0, n1, wt, ct, C):`
	`    if is_reg_empty(R0) and not`
	`    is_reg_empty(R2) and R2[n0_n1]=`
	`    wt and R3[wt]=ct and wt==get_min`
	`    (R3) and C!=0:`
	`        R2[n0_n1]=0`
	`        R3[wt]-ct-1`
	`        C=n0_n1`

6. **Union find of edge endpoints (F_5):** F_5 performs union finding on every extracted edge by F_4.

5.4.2 Complexity Analysis

Kruskal finds the minimum spanning tree in graph G with V nodes and E edges. Using conventional architecture, the complexity of solving Kruskal's algorithm is $\mathcal{O}(E \log_2 E)$ or $\mathcal{O}(E \log_2 V)$. In our representation, each atomic function is associated with different complexity. In the TMMC model, every read and write from memory takes logarithmic time.

The initialization function (F_0) is invoked only one time with constant complexity. The function disjoint set initialization function F_1 makes entry to R_4 and R_5 for every node in $\mathcal{O}(V \log_2 V)$ time. F_{2a} is invoked only for the current node in $\mathcal{O}(\log_2 VE)$ time. F_{2b} is called multiple times to access the rest of the nodes and its connected edge list in $\mathcal{O}(VE \log_2 E)$ time.

Table 5.27

Union find of edge endpoints

F_5: Union find of edge endpoints	F_5: Python Equivalent

Input: n, n', f, f', ct, ct'
Pre: $n \parallel n' \notin R_2$, $R_2 \neq 0$, $C = n \parallel n'$, $n = \mathit{split}(C)[0]$, $n' = \mathit{split}(C)[1]$, $(n, f) \in R_4$, $(n', f') \in R_4$, $f \neq f'$, $n \notin R_6$
Post: $(n \parallel n', f) \in R_6$, $(n', f) \in R_4$, $(f, ct++) \in R_5$, $(f', ct'--) \in R_5$
	`def union_find(n0, n1, f0, f1, ct0, ct1):`
	`s=split(C,'_')`
	`if R2[n0_n1]==0 and is_reg_empty(R2) and C==n0_n1 and n==s[0] and n1==s[1] and R4[n0]==f0 and R4[n1]=f1 and f0!=f1 and R6[n0_n1]==0:`
	`R6[n0]=n1`
	`R4[n1]=f0`
	`R5[f0]=ct0+1`
	`R5[f1]=ct1-1`

F_3 is called until R_0 and R_1 become empty. Reading every edge of every node take $\mathcal{O}(VE\log_2 E)$ time and writing all E edges into R_2 and R_3 takes $\mathcal{O}(E\log_2 E)$ separately. F_4 extracts the minimum edge connected to current node C from R_2 in $\mathcal{O}(E\log_2 E)$ time in total (logarithmic read for the minimum edge until all edges are processed). F_5 function performs union finding in $\mathcal{O}(V\log_2 V)$ time.

5.5 Prim's Algorithm

Prim's algorithm [14] resolves a minimum spanning tree of a graph. It is a greedy algorithm. Using only a adjacency matrix the time complexity of Prim's algorithm gives $\mathcal{O}(V^2)$. Using both a binary heap and an adjacency matrix changes the complexity to $\mathcal{O}(V\log_2 V + E\log_2 V)$.

```
Algorithm 4: Prim $(G, w, r)$
Result: Minimum Spanning Tree
for each $u \in V[G]$ do
    $key[u] \leftarrow \infty$
    $\pi[u] \leftarrow$ NIL
end
$key[r] \leftarrow 0$
$Q \leftarrow V[G]$
for $Q \neq 0$ do
    $u \leftarrow$ EXTRACT-MIN$(Q)$
    for each $v \in Adj[u]$
     do
        if $v \in Q$ and $w(u, v) < key[v]$ then
            $\pi[v] \leftarrow u$
            $key[v] \leftarrow w(u, v)$
        else
            continue
        end
    end
end
```

Figure 5.6

Prim’s algorithm

5.5.1 Predicates and State-transition Functions

Execution of Prim's algorithm to compute single-source shortest path require four OMTs. The roots of these OMTs are stored in five registers (R_0, R_1, R_2, R_3, R_4) along with some additional registers to store constant (C).

- R_0 stores the root of the OMT containing the input graph with the node as key and the nested OMT representing the connected edges as value.
- R_1 stores the root of nested OMT from R_0 containing the neighboring node as key and the edge weight as value.
- R_2 stores the root of OMT containing the node identifiers as key and the path cost from the start node as value.
- R_3 stores the root of OMT containing the edge weights as key and the count of edge weights as value. R_2 and R_3 together form the minimum heap data structure.
- R_4 stores the nodes already visited
- C stores the current node being explored

We need four state-transition functions to execute Prim's algorithm using TMMC.

1. **Initialization (F_0):** This function initializes the root r of input graph OMT to R_0. After the execution of F_0, $R_0 = 0$ will no longer satisfy.
2. **Assigning start node (F_1):** F_1 removes the given start node n from R_0 and assigns it as a current node. The distance of start node n to itself is set to $D = 1$ (as value 0 has a special

Table 5.28

Predicates of Prim's algorithm

Registers	Python Variables
$R_0 : (n, v)$, n=node, v=nested OMT root	`R0 = {}`
$R_1 : (n', v')$, n'=node, v=weight of edge(n, n'), where $n' = neighbor(n)$	`R1 = {}`
$R_2 : (n, w)$, n=node, w=path cost to reach n	`R2 = {}`
$R_3 : (w, ct_w)$, w=edge weight, ct_w=weight count	`R3 = {}`
$R_4 : (n, w)$, n=node visited, w=cost to reach n	`R4 = {}`
C : Stores the current processing node (n) from R_0	`C = 0`

Table 5.29

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0
Pre: $R_0 = R_1 = R_2 = R_3 = R_4 = C = 0$
Post: $R_0 \leftarrow r_0$
	`def init(r0):`
	`    if not R0 and not R1 and not R2`
	`     and not R3 and not R4:`
	`        R0=r0`

interpretation). R_1 stores the value v of n in R_0 representing edges emanating from the current node n.

Table 5.30

Assigning start node

F_1: Assigning start node	F_1: Python Equivalent

Input: n, v, C
Pre: $R_0 \neq 0, R_1 = 0, (n, v) \in R_0, C = 0$
Post: $n \notin R_0$, $R_1 \leftarrow v$, $(n, 1) \in R_2$, $(1, 1) \in R_3$, $C = n$
	`def assign_start_node(n, v, C):`
	`    if R0!=0 and R1!=0 and R0[n]==v`
	`    and C==0 and D==0:`
	`        R0[n]=0`
	`        R1=v`
	`        C=n`

3. **Update node cost (F_2):** F_2 is the function to update the edge cost of nodes in R_2.

4. **Extract minimum weighted edge (F_3):** This function extracts the minimum edge connected to the current node C from R_2 and assigns the adjacent node as the new current node if not already visited.

5.5.2 Complexity Analysis

Prim's algorithm finds the minimum spanning tree of graph G with V nodes and E edges. Using conventional architecture, the complexity of solving Prim's algorithm is $\mathcal{O}(V^2)$; $\mathcal{O}(E \log_2 V)$ if we use adjacency list representation of the graph. In our representation, each atomic function is associated with different complexity.

The initialization function F_0 is invoked only on time in $\mathcal{O}(1)$ time. F_1 removes the given start node n from R_0 and assigns it as a current node. The distance of start node n to itself is set to D

Table 5.31

Update node cost

F_2: Update node cost	F_2: Python Equivalent

Input: n, n', v, ct
Pre: $n \in R_0$, $(n', v) \in R_1$, $n \parallel n' \notin R_2$, $(v, ct) \in R_3$
Post: $n' \notin R_1, (n \parallel n', v) \in R_2, (v, ct+1) \in R_3$
	```
	def update_node_cost(n1, n2, v, ct)
	    :
	    if Ro[n1]!=0 and R1[n2]==v and
	    R2[n1+n2]==0 and R3[v]==ct:
	        R1[n2]=0
	        R2[n1+n2]=v
	        R3[v]=ct
	```

Table 5.32

Extract minimum weighted edge, case (a), node not visited

F_{3a}: Extract minimum weighted edge, case (a)	F_{3a}: Python Equivalent

Input: n', v, w, ct, C
Pre: $R_0 \neq 0$, $C \in R_0$, $(n', v) \in R_0$, $R_1 = 0$, $(n \parallel n', w) \in R_2$, $(w, ct) \in R_3$, $w = get\_min(R_3)$, $n' \notin R_4$
Post: $C \notin R_0$, $C \leftarrow n'$, $R_1 \leftarrow v$, $n \parallel n' \notin R_2$, $(w, ct-1) \in R_3$, $(n', w) \in R_4$
	```
	def extract_min_edge_a(n1, n2, v, w
	    , ct, C):
	    if not R0 and R0[C]!=0 and R0[
	    n2]=v and not R1 and R2[n1+n2]==
	    w and R3[w]==ct and w==get_min(
	    R3) and R4[n2]==0:
	        R0[C]=0
	        C=n2
	        R1=v
	        R2[n1+n2]=0
	        R3[w]=ct-1
	        R4[n2]=w
	```

Table 5.33

Extract minimum weighted edge, case (b), node already visited

F_{3b}: Extract minimum weighted edge, case (b)	F_{3b}: Python Equivalent

Input: n', v, w, ct
Pre: $R_0 \neq 0$, $C \in R_0$, $(n', v) \in R_0$, $R_1 = 0$, $(n \parallel n', w) \in R_2$, $(w, ct) \in R_3$, $w = get\_min(R_3)$, $n' \in R_4$
Post: $n \parallel n' \notin R_2, (w, ct-1) \in R_3$
	```
	def extract_min_edge_b(n1,n2,v,w,ct
	    ):
	    if not R0 and R0[C]!=0 and R0[
	    n2]=v and not R1 and R2[n1+n2]==
	    w and R3[w]==ct and w==get_min(
	    R3) and R4[n2]!=0:
	        R2[n1+n2]=0
	        R3[w]=ct-1
	```

= 1 (as value 0 has a special interpretation). R_1 stores the value v of n in R_0 representing edges emanating from the current node n. It is also invoked one time ($\mathcal{O}(1)$).

F_2 is the function to update the path cost of nodes in R_2. This function is invoked VE times in total and in each invocation the edge read takes $\mathcal{O}(\log_2 E)$ as opposed to the conventional constant time reading.

F_3 extracts the minimum edge connected to the current node C from R_2 and assigns the adjacent node as the new current node if not already exists in R_4 (if not already visited). In TMMC, extracting maximum and minimum from the OMT takes logarithmic time. So here this function extracts minimum edge for all the nodes is ($\mathcal{O}(V \log_2 E)$.

CHAPTER VI

COMPUTATIONAL GEOMETRY ALGORITHMS

Computational geometry is one of the oldest areas of scientific research. It contains the field of algorithms that involves geometry. Several computational geometry algorithms exist for solving different computational geometry problems. A large portion of these algorithms involves proving several geometrical properties. Computational geometry algorithms such as convex hull, partitioning a polygon into monotone pieces, half-plane intersection, point location, n-closest points, etc., have compelling real-world applications in computer graphics, robotics, computer-aided design, graphs, geographic information systems, pattern recognition, molecular modeling, etc.

We look at three such algorithms in this section. Our objective is to ensure the correctness of the execution of these algorithms using the proposed TMMC (TCB Minimizing Model of Computation).

6.1 Convex Hull

Convex Hull is one of the most commonly used computational geometry problems. It is widely used in the field of computer visualization (Example: video games, robotics), Geographical Information Systems (GIS) (Example: Maps), image processing, pattern recognition (example: bounding boxes detection), etc. Some of the practical applications of finding the convex hull involve collision detection of vehicles, finding the area of the disease epidemic, motion planning of robots, selection of a particular area in images for processing, bounding boxes detection, etc.

A subset S of the plane is called convex if and only if for any pair of points $p, q \in S$ (Figure 6.1) the line segment ($\overline{pq}$) is completely contained in S. The convex hull CH of a set S is the smallest number convex set that contains S. To be more precise, it is the intersection of all convex sets that contains S.

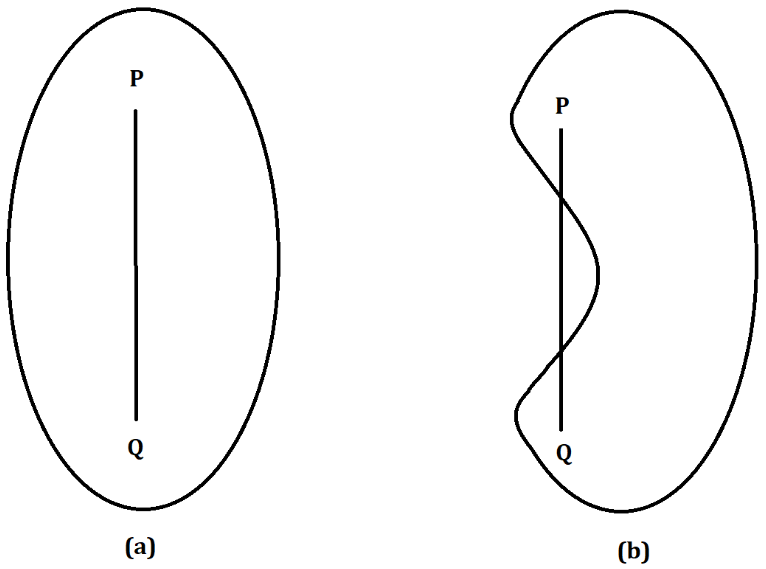


Figure 6.1

(a) Convex (b) Not convex

6.1.1 Graham's Scan Algorithm

Convex Hull for a set of points can be calculated using various algorithms. One of the popular ones is Graham's scan algorithm (Figure 6.2) [15]. The problem of finding the convex hull for a set of points is divided into finding the upper convex hull and the lower convex hull (Figure 6.3) and further merging them to give the final convex hull for all the points. The complexity of this algorithm is $\mathcal{O}(N \log_2 N)$ (N = the number of points in the set) using the conventional model of computation.

6.1.2 Predicates and State-transition Functions

We will use three registers to track the roots of the four OMTs needed to solve the Convex Hull problem and four registers to store constant (P_{c-1}, P_c, P_{c+1} and P_{c+2}).

- R_0 stores the input points. Key of R_0 is the point identifier p and value is the x, y coordinate concatenated together i.e. $(p, x||y)$
- R_1 stores $x||y$ as key and point identifier p as value. The key here is in ascending order i.e. $(x||y, p)$
- R_2 Hold the copy of R_1
- P_{c-1}, P_c, P_{c+1} and P_{c+2} hold points in R_1 and R_2 to detect left or right turns.

To solve the convex hull problem using the proposed TMMC model, we describe the Grahams scan's Algorithm by four state-change functions (F_1, F_2, F_3, F_4).

1. **Initialization (F_0):** Invocation of this state-change function assigns the OMT root of the input points and sorted x coordinates. It adds all the input points to R_0 with a unique identifier p and coordinates x and y i.e. $(p, x||y)$. It also populates R_1 with the x-coordinator concatenated with its corresponding y coordinate as keys and point identifier as value (this ensures that every point has a unique identifier) i.e. $(x||y, p)$

Algorithm 5: Graham's Scan Algorithm (P)

```
Result: A list of vertices in clockwise order representing $CH(P)$
Sort the points by $x$-coordinate, resulting in a sequence $p_1, \ldots, p_k$
Insert $p_1$ and $p_2$ in list $L_{upper}$, with $p_1$ as the first point
for $i \leftarrow 3$ to $k$ do
    Append $p_i$ to $L_{upper}$
    $n = length(L_{upper})$
    while $n > 2$ and the $p_{n-2}, p_{n-1}, p_n \in L_{upper}$ do not make a right turn do
        Delete $p_{n-1}$ from $L_{upper}$
    end
end
Insert $p_k$ and $p_{k-1}$ in list $L_{lower}$, with $p_k$ as the first point
for $i \leftarrow k-2$ down to 1 do
    Append $p_i$ to $L_{lower}$
    $m = length(L_{lower})$
    while $m > 3$ and $p_{m-2}, p_{m-1}, p_m \in L_{lower}$ do not make a right turn do
        Delete $p_{m-1}$ from $L_{lower}$
    end
end
Remove $p_0$ and $p_m$ from $L_{lower}$ to avoid duplication in the the upper and lower hull
Append $L_{lower}$ to $L_{upper}$, and call the resulting list $L$
return $L$
```

Figure 6.2

Graham's scan algorithm

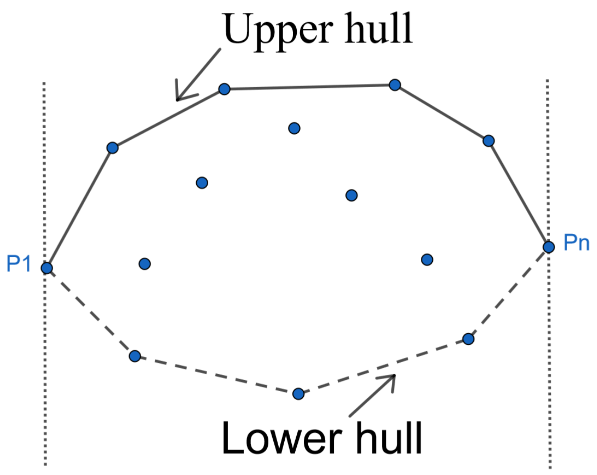


Figure 6.3

Upper and lower convex hull computation

Table 6.1

Predicates of Grahams scan's algorithm

Registers	Python Variables

R_0: $(p, x \parallel y)$, p=point id, $x \parallel y$=coordinates
R_1: $(x \parallel y, p)$, $x \parallel y$=coordinates, p=point id (upper)
R_2: $(x \parallel y, p)$, $x \parallel y$=coordinates, p=point id (lower)
P_{c-1}: Stores previous key of the current point
P_c: Stores current point
P_{c+1}: Stores next key of P_c
P_{c+2}: Stores next key of P_{c+1}
	`R0={}`
	`R1={}`
	`R2={}`
	`Pc0=Pc1=Pc2=Pc3=0`

After all inputs have been provided, this state-change function

(a) creates a third OMT (R_2) which is identical to R_1. R_1 will be altered when the upper convex hull points are determined. R_2 will be altered when lower convex hull points are determined.

(b) sets a state *current-point* to the least index in R_1 and this current point is stored in P_c. The value of P_c is copied to P_{c-1} and P_{c+1} and P_{c+2} are assigned the next two lowest keys of P_c in R_1

2. **Detect turns for upper convex hull (F_1):** This state-change function begins with the *current-point* in P_c and depending on the next two points, either advance the current point to the next point or drops the next point.

   It takes into account three points at a time (the current point P_c, its next point P_{c+1} and the next of its next point P_{c+2}). These three points are used to detect a left or a right turn. The turn is detected by calculating the determinant of these three points. The three points may lie on a straight line which is equivalent to a right turn for the upper CH. In case of a right

Table 6.2

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0, r_1
Pre: $R_0 = R_1 = R_2 = 0, P_{c-1} = P_c = P_{c+1} = P_{c+2} = 0, length(r_0) = length(r_1)$
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $R_2 \leftarrow r_1$, $P_c \leftarrow get\_min(R_1)$, $P_{c-1} \leftarrow P_c$, $P_{c+1} \leftarrow next\_key(P_c, R_1)$, $P_{c+2} \leftarrow next\_key(P_{c+1}, R_1)$
	`def init(r0, r1):`
	`    if not R0 and not R1 and not R2`
	`     and Pc0==Pc1==Pc2==Pc3==0:`
	`        R0=r0`
	`        R1=r1`
	`        R2=r1`
	`        Pc1=get_min(R_1)`
	`        Pc0=Pc1`
	`        Pc2=next_key(Pc1, R1)`
	`        Pc3=next_key(Pc2, R1)`

turn, all three points are advanced to the next point (their next key in R_1). In case of a left turn, P_{c+1} and P_{c+2} are advanced to the next points.

3. **Initialize lower convex hull (F_2):** At any point when the next point of P_{c+2} has a lower index (which signifies that the walk-through is complete), P_c, P_{c+1}, P_{c+2} are set again to the smallest, next smallest and next-next smallest index of R_2 to begin lower-hull computation.

4. **Detect turns for lower convex hull (F_3):** Similar to F_1 state-change function, this function begins with the *current-point* P_c in R_2 and depending on the next two points, either advance the current point to the next point or drops the next point.

   It takes into account three points at a time (the current point P_c, its next point P_{c+1} and the next of its next point P_{c+2}). These three points are used to detect a left or a right turn. The turn is detected by calculating the determinant of these three points. The three points may lie on a straight line which is equivalent to a left turn. In the case of a left turn, all three points

Table 6.3

Detect turns for Upper CH: case (a) (right turn)

F_{1a}: Detect turns for upper convex hull, case (a)	F_{1a}: Python Equivalent

Input: P_c, P_{c+1}, P_{c+2}
Pre: $length(R_1) \geq 3$, $\{P_c, P_{c+1}, P_{c+2}\} \in R_0$, $determinant(P_c, P_{c+1}, P_{c+2}) \geq 0$
Post: $P_c = P_{c+1}$, $P_{c+1} = P_{c+2}$, $P_{c+2} = next\_key(P_{c+2}, R_1)$
	`def detect_turn_upper_a(Pc1, Pc2,`
	`    Pc3):`
	`     if length(R1)>=3 and R0[Pc1]==`
	`    R1[Pc1] and R0[Pc2]==R1[Pc2] and`
	`     R0[Pc3]==R1[Pc3] and`
	`    determinant(Pc1, Pc2, Pc3)>=0:`
	`         Pc1=Pc2`
	`         Pc2=Pc3`
	`         Pc3=next_key(Pc3, R1)`

Table 6.4

Detect turns for Upper CH: case (b) (left-left turn)

F_{1b}: Detect turns for upper convex hull, case (b)	F_{1b}: Python Equivalent

Input: $P_{c-1}, P_c, P_{c+1}, P_{c+2}$
Pre: $length(R_1) \geq 3$, $\{P_{c-1}, P_c, P_{c+1}, P_{c+2}\} \in R_0$, $determinant(P_c, P_{c+1}, P_{c+2}) < 0$, $determinant(P_{c-1}, P_c, P_{c+2}) < 0$
Post: $P_c = P_{c-1}, P_{c+1} = P_c, P_{c+1} \notin R_1$
	`def detect_turn_upper_b(Pc0, Pc1,`
	`    Pc2, Pc3):`
	`     if length(R1)>=3 and R0[Pc0]==`
	`    R1[Pc0] and R0[Pc1]==R1[Pc1] and`
	`     R0[Pc2]==R1[Pc2] and R0[Pc3]==`
	`    R1[Pc3] and determinant(Pc1, Pc2`
	`    , Pc3)<0 and determinant(Pc0,`
	`    Pc1, Pc2)<0:`
	`         Pc1=Pc0`
	`         Pc2=Pc1`
	`         R1[pc2]=0`

Table 6.5

Detect turns for Upper CH: case (c) (left-right turn)

F_{1c}: Detect turns for upper convex hull, case (c)	F_{1c}: Python Equivalent

Input: $P_{c-1}, P_c, P_{c+1}, P_{c+2}$
Pre: $length(R_1) \geq 3$, $\{P_{c-1}, P_c, P_{c+1}, P_{c+2}\} \in R_0$, $determinant(P_c, P_{c+1}, P_{c+2}) < 0$, $determinant(P_{c-1}, P_c, P_{c+2}) \geq 0$
Post: $P_{c+1} = P_{c+2}$, $P_{c+2} = next\_key(P_{c+2}, R_1)$, $P_{c+1} \notin R_1$
	`def detect_turn_upper_c(Pc0, Pc1, Pc2, Pc3):`
	    `if length(R1)>=3 and R0[Pc0]==R1[Pc0] and R0[Pc1]==R1[Pc1] and R0[Pc2]==R1[Pc2] and R0[Pc3]==R1[Pc3] and determinant(Pc1, Pc2, Pc3)<0 and determinant(Pc0, Pc1, Pc3)>=0:`
	        `Pc2=Pc3`
	        `Pc3=next_key(Pc3, R3)`
	        `R1[Pc2]=0`

Table 6.6

Lower CH initialization

F_2: Initialization for lower convex hull	F_2: Python Equivalent

Input: $P_{c-1}, P_c, P_{c+1}, P_{c+2}$
Pre: $next\_key(P_{c+2}, R_1) < P_{c+2}$
Post: $P_c \leftarrow get\_min(R_2)$, $P_{c-1} \leftarrow P_c$, $P_{c+1} \leftarrow next\_key(P_c, R_2)$, $P_{c+2} \leftarrow next\_key(P_{c+1}, R_2)$
	`def init_lower(Pc0, Pc1, Pc2, Pc3):`
	    `if next_key(Pc3, R1) < Pc3:`
	        `Pc1=get_min(R_2)`
	        `Pc0=Pc1`
	        `Pc2=next_key(Pc1, R2)`
	        `Pc3=next_key(Pc2, R2)`

are advanced to the next point (their next key in R_2). In case of a right turn, P_{c+1} and P_{c+2} are advanced to the next points.

Table 6.7

Detect turns for Lower CH: case (a) (left turn)

F_{3a}: Detect turns for lower convex hull, case (a)	F_{3a}: Python Equivalent

Input: P_c, P_{c+1}, P_{c+2}
Pre: $length(R_1) \geq 3$, $\{P_c, P_{c+1}, P_{c+2}\} \in R_0$, $determinant(P_c, P_{c+1}, P_{c+2}) \leq 0$
Post: $P_c = P_{c+1}$, $P_{c+1} = P_{c+2}$, $P_{c+2} = next\_key(P_{c+2}, R_1)$
	`def detect_turn_lower_a(Pc1, Pc2, Pc3):`
	`  if length(R1)>=3 and R0[Pc1]== R1[Pc1] and R0[Pc2]==R1[Pc2] and R0[Pc3]==R1[Pc3] and determinant(Pc1, Pc2, Pc3)<=0:`
	`    Pc1=Pc2`
	`    Pc2=Pc3`
	`    Pc3=next_key(Pc3, R1)`

5. **Merge point from lower to upper convex hull (F_4):** At any point duirng F_3, when the next point of P_{c+2} has a lower index (which signifies that the walk-through is complete) we move on to F_4, the merge function. F_4 state-change function merges the lower convex hull R_2 with the upper convex hull in R_1. This is achieved by copying all the entries of R_2 to R_1. F_4 copies each key-value pair of R_2 tp R_1 only if it does not already exist in R_1. This results in the final convex hull solution in R_1.

6.1.3 Complexity Analysis

Graham's scan algorithm finds the convex hull of the input point set P. Using conventional architecture, the complexity of solving Graham's scan algorithm is $\mathcal{O}(P \log_2 P)$. In our represen-

Table 6.8

Detect turns for Lower CH: case (b) (right-right turn)

F_{3b}: Detect turns for lower convex hull, case (b)	F_{3b}: Python Equivalent

Input: $P_{c-1}, P_c, P_{c+1}, P_{c+2}$
Pre: $length(R_1) \geq 3$, $\{P_{c-1}, P_c, P_{c+1}, P_{c+2}\} \in R_0$, $determinant(P_c, P_{c+1}, P_{c+2}) > 0$, $determinant(P_{c-1}, P_c, P_{c+2}) > 0$
Post: $P_c = P_{c-1}, P_{c+1} = P_c, P_{c+1} \notin R_1$
	`def detect_turn_lower_b(Pc0, Pc1,`
	`Pc2, Pc3):`
	`if length(R1)>=3 and R0[Pc0]==`
	`R1[Pc0] and R0[Pc1]==R1[Pc1] and`
	`R0[Pc2]==R1[Pc2] and R0[Pc3]==`
	`R1[Pc3] and determinant(Pc1, Pc2`
	`, Pc3)>0 and determinant(Pc0,`
	`Pc1, Pc2)>0:`
	`Pc1=Pc0`
	`Pc2=Pc1`
	`R1[pc2]=0`

Table 6.9

Detect turns for Lower CH: case (c) (right-left turn)

F_{3c}: Detect turns for lower convex hull, case (c)	F_{3c}: Python Equivalent

Input: $P_{c-1}, P_c, P_{c+1}, P_{c+2}$
Pre: $length(R_1) \geq 3$, $\{P_{c-1}, P_c, P_{c+1}, P_{c+2}\} \in R_0$, $determinant(P_c, P_{c+1}, P_{c+2}) > 0$, $determinant(P_{c-1}, P_c, P_{c+2}) \leq 0$
Post: $P_{c+1} = P_{c+2}$, $P_{c+2} = next\_key(P_{c+2}, R_1)$, $P_{c+1} \notin R_1$
	`def detect_turn_lower_vc(Pc0, Pc1,`
	`Pc2, Pc3):`
	`if length(R1)>=3 and R0[Pc0]==`
	`R1[Pc0] and R0[Pc1]==R1[Pc1] and`
	`R0[Pc2]==R1[Pc2] and R0[Pc3]==`
	`R1[Pc3] and determinant(Pc1, Pc2`
	`, Pc3)>0 and determinant(Pc0,`
	`Pc1, Pc3)<=0:`
	`Pc2=Pc3`
	`Pc3=next_key(Pc3, R3)`
	`R1[Pc2]=0`

Table 6.10

Merge

F_4: Merge point from lower to upper convex hull	F_4: Python Equivalent

Input: pid, v
Pre: $(pid, v) \in R_4, pid \notin R_3, length(R_2) < 3$
Post: $pid \notin R_4, pid \in R_3$
	`def merge_upper_lower(pid, v):`
	`if R4[pid]==v and R3[pid]==0`
	`and length(R2)<3:`
	`R4[pid]=0`
	`R3[pid]=v`

tation, each atomic function is associated with different complexity. In the TMMC model, every read and write from memory takes logarithmic time.

The initialization function F_0 is invoked only once (constant time). It assign some values ot the registers. The $get\_min()$ and $next\_key()$ function takes $\mathcal{O}(\log_2 P)$ time. F_2 function also function a few assignments to the registers.

F_1 and F_3 functions updates R_1 and R_2, P times and each update takes ($\mathcal{O}(\log_2 P)$) time. The utility function used here ($determinant()$) has its separate complexity but it can be readily executed separately. F_4 merges R_1 into R_2 in $\mathcal{O}(P \log_2 P)$ time.

6.2 Line Segments Intersection Detection

Line segment intersection problem is a fundamental problem in computational geometry. Bentley and Ottoman [8] algorithms provide a solution to this problem where the algorithm finds all the intersections in $\mathcal{O}(N log_2 N)$ (N = number of segments) time. Another algorithm introduced before the Bentley and Ottoman one was proposed by Shamos and Hoey [59]. The Shamos and Hoey algorithm can be considered as the simpler version of the Bentley and Ottmann algorithm that detects if the given line segments have at least one intersecting point or not in $\mathcal{O}(N \log_2 N)$ time. This algorithm does not necessarily compute the intersecting points.

Detecting the intersection of a set of line segments is an important geometrical problem. A polygon is considered to be simple if no edges of this polygon have a crossover with one another. Shamos and Hoey algorithm can thus be used to detect whether a polygon is a simple polygon. The sweep/scan line algorithm [59] technique was used by Shamos *et al* in their algorithm along with an efficient self-balancing binary tree data structure.

Edges without any crossover give a planar graph which is highly desirable in electrical circuit design so as in computer graphics and geographical information system where objects are rendered visible or obscure based their layered structure.

6.2.1 Shomos-Hoey Algorithm

In the Sweep line algorithm by Shamos and Hoey [59] to detect intersection in a given set of line segment considers a conceptual vertical line from left to right (Figure 6.4). At first, the endpoints of the line segments are sorted and the vertical sweep line is considered which scans these endpoints from left to right. Once the left endpoint is encountered it is inserted into the self-balancing binary tree and once the corresponding right endpoint is encountered we delete the left endpoint. Any time if an attempt is made to insert a left endpoint when the data structure is empty we know an overlap has occurred.

6.2.2 Predicates and State-transition Functions

We need three OMTs to execute the Shamos-Hoey algorithm. These three OMTs are stored in three registers. One more register is used to store a constant.

- R_0 stores line segments described by s, x_1, y_1, x_2, y_2 as $(s, x_1 \parallel y_1 \parallel x_2 \parallel y_2)$ where s is a line segment identifier (key), (x_1, y_1) and (x_2, y_2) are its two end points, where $x_1 < x_2$ (if $x_1 = x_2$ then $y_1 < y_2$).
- R_1 has two entries for each line segment in R_0 of the form $(x_1 \parallel y_1 \parallel s, s)$. The key $x_1 \parallel y_1 \parallel S$ represents one of the endpoints of s.

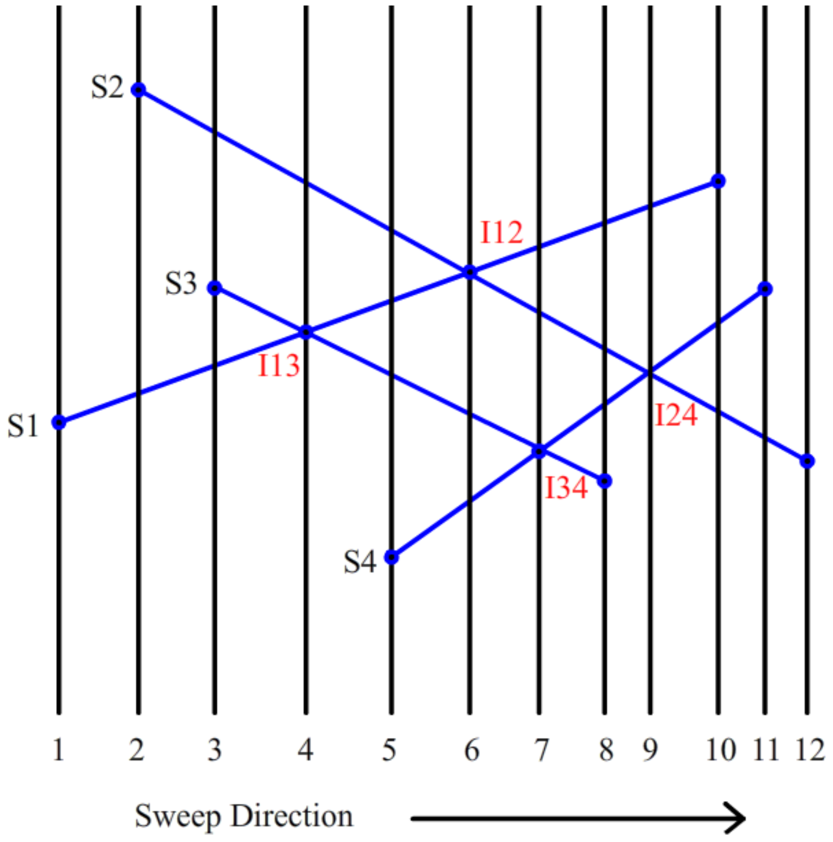


Figure 6.4

Intersections of a set of segments

```
Algorithm 6: Shamos-Hoey($S_1, S_2, \ldots, S_n$)
  Result: Existence of intersection in the input line segments
  Sort the endpoints lexicographically on $x$ and $y$ coordinates such that
  $point[1]$ = leftmost
  $point[2N]$ = rightmost
  for $i$=1 to 2N do
      $p = point[i]$
      Let $S$ = segment of which $p$ is an endpoint
      if $p$ = left_endpoint($S$) then
          INSERT($S, T$)
          A:=ABOVE($S, T$)
          B:=BELOW($S, T$)
          if $A$ intersects $S$ then
              RETURN($A, S$)
          end
          if $B$ intersects $S$ then
              RETURN($B, S$)
          end
      end
      else if $p$=right_endpoint($S$) then
          A:=ABOVE($S, T$)
          B:=BELOW($S, T$)
          if $A$ intersects $B$ then
              RETURN($A, B$)
          end
          DELETE($S, T$)
      end
  end
  INSERT($s, T$) inserts the segments into the total order maintained by $T$.
  DELETE($s, T$) deletes segments.
  ABOVE($s, T$) returns the name of the segment immediately aboves in $T$
  BELOW($s, T$) returns the name of the segment immediately below $S$.
```

Figure 6.5

Shomos-Hoey Algorithm

- R_2 stores the y-coordinates. It has the format $(y \parallel s, s)$ where $y \parallel s$ is a unique identifier for the y coordinate of segment s. R_3 only contains start point's y coordinate therefore the key will always be unique.
- C stores the lowest key in R_1
- Y stores the y coordinate of C
- S stores the segment identifier of C

Table 6.11

Predicates of Shamos-Hoeys algorithm

Registers	Python Variables

R_0 : $(s,\ x_1 \parallel y_1 \parallel x_2 \parallel y_2)$, s=segment id, x_1, y_1, x_2, y_2=coordinates
R_1 : $(x \parallel y \parallel s, s)$, x, y=coordinate, s=Segment id
R_2 : $(y \parallel s, s)$, y=coordinate, s=Segment id
C : current minimum endpoint in R_1
Y : y coordinate of C
S : Segment id of C
	`R0 = {}`
	`R1 = {}`
	`R2 = {}`
	`C=0`
	`Y=0`
	`S=0`

To detect line segment intersections using the proposed TMMC model, we describe Shamos-Hoey's Algorithm by three state-change functions (F_1, F_2, F_3).

1. **Initialization (F_0):** The state-change function assigns the root of OMT to R_0 containing the line segments described by S, x_1, y_1, x_2, y_2 where S is a line segment identifier, (x_1, y_1) and (x_2, y_2) are its two end points, where $x_1 < x_2$ (if $x_1 = x_2$ then $y_1 < y_2$).

   This function also populates R_1 with coordinates (sorted inherently) as keys of the form $(x_1 \parallel y_1 \parallel S)$.

On completion, the keys in R_1 are ordered by increasing the value of x coordinates. Appending y coordinate and the line segment identifier ensures the uniqueness of keys. On completion of all inputs, the start-point, C is set to the lowest index in R_1.

Table 6.12

Initialization

F_0: Initialization	F_0: Python Equivalent
	`def init(r0, r1, C, Y):`
	`    if not R0 and not R1 and not R2`
Input: r_0, r_1, C, Y	`    and C==Y==S==0:`
Pre: $R_0 = R_1 = R_2 = C = Y = S = 0$	`        R0 = r0`
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $C \leftarrow extract\_min(R_1)$,	`        R1 = r1`
$Y \leftarrow C.split()[1]$, $S \leftarrow C.split()[2]$, $(Y \parallel S, S) \in$	`        C=extract_min(R1),`
R_2	`        Y=C.split()[1]`
	`        S=C.split()[2]`
	`        R2[Y+","+S]=S`

2. **Insert y coordinate in case of a left endpoint (F_1):** F_1 state-change function extracts the lowest index (minimum x) of R_1 and inserts its corresponding y coordinate into an OMT (R_2) if the extracted point is a left endpoint. In R_2 the index is ($y \parallel S$) where S is the line segment identifier appended with the y coordinate to ensure uniqueness and the value is the line segment identifier S. F_2 checks for intersection between the line segment S and the segment above it (A) in R_2 (previous index of $y \parallel S$). It also checks for intersection between the line segment S and the segment below it (B) in R_2 (next index of $y \parallel S$). Ideally there should be no intersections.

   F_3 checks whether the extracted minimum x belongs to a left endpoint. This basic functionality just performs a few register operations in constant time and check the intersection of segments in case of a left endpoint. There are things to be checked, one the corresponding segment of the y coordinate and the segment above it do not intersect and two the corresponding segment of the y coordinate and the segment below it do not intersect.

Table 6.13

Remove y coordinate in case of a right endpoint

F_1: Insert y coordinate in case of a left endpoint	F_1: Python Equivalent

Input: a, b, C, Y, S
Pre: $next\_key(C, R_0) > C$, $(S, v) \in R_0$, $C.split()[0] < v.split()[2]$, $a = above(S, R_2)$, $b = below(S, R_2)$, $does\_not\_intersect(a, S)$, $does\_not\_intersect(b, S)$
Post: $C \leftarrow extract\_min(R_1)$, $Y \leftarrow C.split()[1]$, $S \leftarrow C.split()[2]$, $(Y \parallel S, S) \in R_2$
	`def insert_y_left(a, b, C, Y, S):`
	`    if next_key(C, R0) > C and R0[S]==v and C.split(',')[0] < v.split(',')[2] and a==above(S, R2) and b==below(S, R2) and does_not_intersect(a,S) and does_not_intersect(b,S):`
	`        C = extract_min(R1)`
	`        Y = C.split(',')[1]`
	`        S = C.split(',')[2]`
	`        R2[Y+','+S]=S`

3. **Detecting intersection (right endpoint) (F_2):** If the extracted point is a right endpoint and belongs to line segment S, F_2 checks for the intersection between the above line segment A and the below line segment B of S. Ideally, there should be no intersections. This state-change function finishes completion when the content of R_1 is empty. F_5 checks whether the extracted minimum x belongs to a right endpoint.checks intersection of segments in case of a right endpoint. It checks the segment above and below the corresponding segment do not intersect. When all the entries of R_1 are considered, the algorithm concludes.

Table 6.14

Detecting intersection (right endpoint)

F_2: Remove y coordinate in case of a right endpoint	F_2: Python Equivalent
Input: a, b, C, Y, S Pre: $next\_key(C, R_0) > C$, $(S, v) \in R_0$, $C.split()[0] > v.split()[0]$, $a = above(S, R_2)$, $b = below(S, R_2)$, $does\_not\_intersect(a, b)$ Post: $C \leftarrow extract\_min(R_1)$, $Y \leftarrow C.split()[1]$, $S \leftarrow C.split()[2]$, $Y \parallel S \notin R_2$	`def remove_y_right(a, b, C, Y, S):` `if next_key(C, R0) > C and R0[S]==v and C.split(',')[0] > v.split(',')[0] and a==above(S, R2) and b==below(S, R2) and does_not_intersect(a,b):` `C = extract_min(R1)` `Y = C.split(',')[1]` `S = C.split(',')[2]` `R2[Y+','+S]=0`

6.2.3 Complexity Analysis

Shamos Hoey's finds whether any of the two line segments in a given set of P line segments intersect each other or not. Using conventional architecture, the complexity of solving Shamos-Hoey's algorithm is $\mathcal{O}(N \log_2 E)$. In our representation, each atomic function is associated with different complexity. In the TMMC model, every read and write from memory takes logarithmic time.

The initialization function F_0 is invoked only once (constant time). It assigns two OMT roots to R_0 and R_1 in constant time. It also calls $extract\_min()$ function which takes $\mathcal{O}(\log_2 P)$ time.

F_1 function checks for intersection only by checking the above and below segments in R_2 and it takes $\mathcal{O}(\log_2 P)$ time. It also extracts minimum x coordinate from R_0 in $\mathcal{O}(\log_2 P)$ time. F_1 is repeated P times for all the left endpoints. F_2 similar to F_1 is also perform within same time complexity for all the right endpoints. These two functions also use a utility function $does\_not\_intersect()$ and $split()$ that have their separate complexity not exceeding $\mathcal{O}(\log_2 P)$.

6.3 Polygon Triangulation

Triangulation is an important technique used to determine the location of a point on a plane. There are numerous ways to triangulate a set of points (or a polygon) (Figure 6.6). Delaunay triangulation is one of the most common types of triangulation invented by a Russian mathematician, Boris Delaunay in 1934 [16].

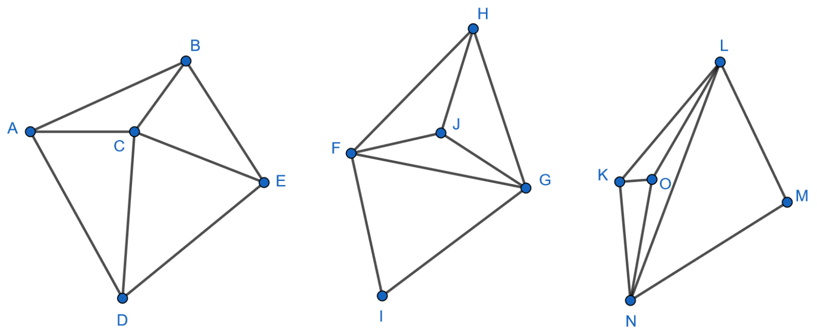


Figure 6.6

Three sets of triangulation for a given set of five points

6.3.1 Delaunay Triangulation

Delaunay triangulation of a point set P creates a triangular mesh where for each triangle no other points in P reside within the circumcircle of that triangle. Delaunay triangulation considers no three points are co-linear and avoids triangles with narrow angles. Another concept introduced earlier than Delaunay triangulation is Voronoi diagram [66] (Figure 6.7: blue edges). Delaunay triangulation is the dual graph of the Voronoi Diagram. Voronoi Diagram can be computed from

Delaunay triangulation or directly using the sweepline technique (e.g. Fortune's Algorithm [20] $\mathcal{O}(P \log_2 P)$).

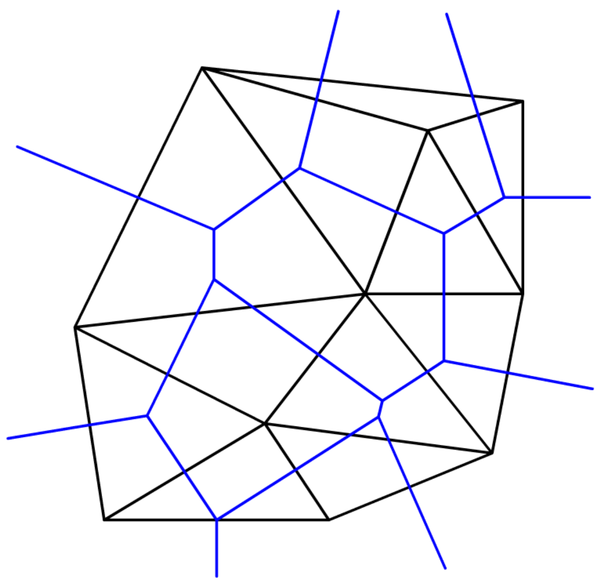


Figure 6.7

Delaunay triangulation (black) and Voronoi diagram (blue)

It is possible to perform Delaunay triangulation of a given set of points easily with the help of a wide range of available algorithms. Couple of the algorithms to mention here are Flip algorithm ($\mathcal{O}(P^2)$ edge flips) [15], incremental algorithm ($\mathcal{O}(P^2)$)) (e.g. Bowyer-Watson algorithm [10]), Divide and Conquer algorithm by Guibas and Stolfi ($\mathcal{O}(P \log_2 P)$)[25] etc.

Delaunay triangulation has numerous applications especially in the field of computer graphics e.g. terrain generation, reconstruction, meshing, etc. The graph representing Delaunay triangulation also can be referred to as the nearest neighbor graph to compute n-nearest neighbors of a point of interest in a 2D plane. Delaunay triangulation is also used for path planning which is useful in robotics.

The dual graph of Delaunay triangulation i.e. Voronoi diagram is related to a range of other significant problems such as a k-nearest neighbor, minimum spanning tree, largest empty circle, smallest enclosing circle, Gabriel graph and so on [58]. Delaunay triangulation is also related to Convex Hull Problem discussed in the previous section. The exterior facing boundary of a Delaunay triangulation of a set of points P can be also called the convex hull for the same point set P (Figure 6.7: Blue edges). Voronoi Diagram is extensively used to understand patterns exist in nature. The major feature of this type of triangulation is to be able to deduce the location of points. Delaunay triangulation can also be used to find N-nearest neighbor of a certain point in the 2D plane version as the nearest neighbor graph is the subset of Delaunay triangulation.

6.3.2 Verification of Delaunay Triangulation Solution

Our major goal for the proposed TMMC model is to make the verification steps as simple as possible. Due to the complex nature of the algorithms that solve the Delaunay triangulation problem, We propose a verification algorithm that verifies the correctness of the given Delaunay triangulation meaning verifying whether the set of points was correctly triangulated. To verify that a given Delaunay triangulation's correctness is a bit tricky. Proving the circumcircle property won't suffice. Suppose we are given a Delaunay triangulation T with a set of Edges E and points P. We have to ensure the following properties:

1. Subset of the edges ($E' \subseteq E$) of the triangulation T must form a convex hull (CH) for all the points P (Figure 6.8). In Figure 6.8(b) edges (A_1, B_1), (B_1, F_1), (F_1, E_1), (E_1, D_1) and

(D_1, A_1) form a convex hull for points $A_1, B_1, C_1, D_1, E_1, F_1$ whereas in Figure 6.8(a) the edges do not form a Convex Hull.

2. The Delaunay triangulation T must follow the maximal planar subdivision property meaning the triangulation should be a planar graph. No edge of the triangulation can cross each other. Figure 6.9 is an improper Delaunay triangulation.

3. Circumcircle of each of the triangles must not contain any other point of any other triangles (should be an empty circle) (Figure 6.10).

4. The given Delaunay triangulation T should not be isolated. For example Figure 6.11

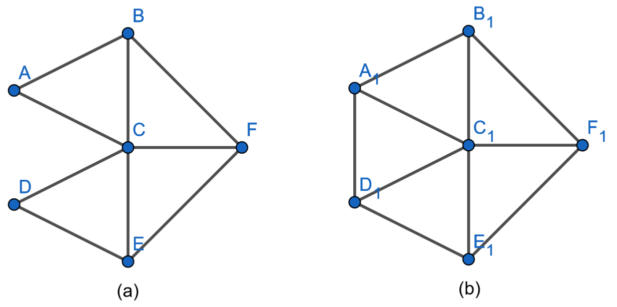


Figure 6.8

(a) Improper Delaunay triangulation, (b) Proper Delaunay triangulation

Given the Delaunay triangulation and its points, we can securely execute the Convex Hull algorithm using TMMC to verify the convex hull property (property 1 above). In addition to that, property 2 can also be verified in the same way by executing the Shamos-Hoey algorithm that helps us detect if there is any edge crossover within the given Delaunay triangulation. In this section,

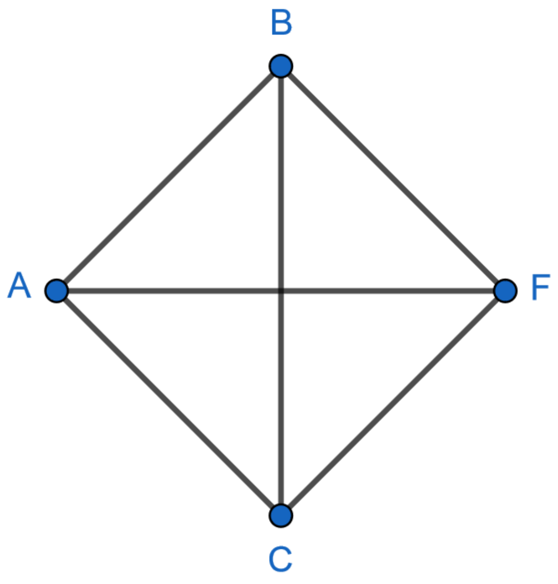


Figure 6.9

Non-planar triangulation. Edge AF and BC cross each other

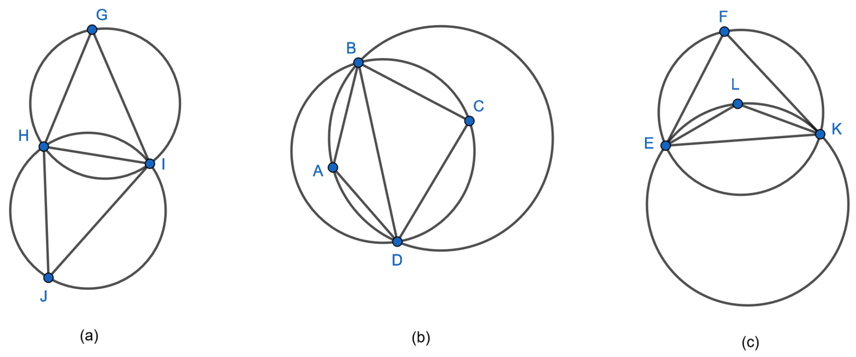


Figure 6.10

Circumcircle property (a) is a proper Delaunay triangulation, (b) and (c) are not

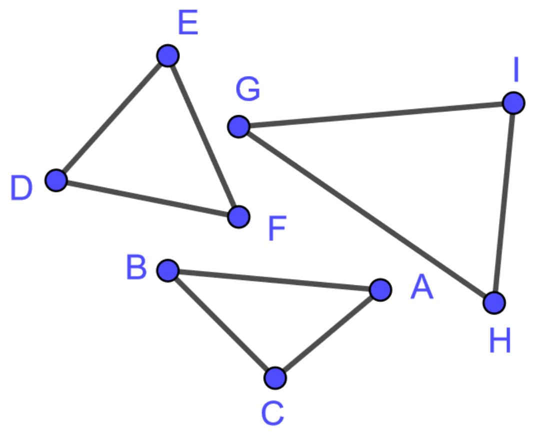


Figure 6.11

Isolated triangulation of the set of points $\{A, B, C, D, E, F, G, H, I\}$

we verify the correctness of the given Delaunay triangulation solution by checking the primary circumcircle property (Figure 6.10) along with the connectivity of the triangles (Figure 6.11) property through union-find. We first provide the corresponding verification algorithm in algorithm (Figure 6.12) and then describe the atomic state-changes corresponding to the algorithm.

6.3.3 Predicates and State-transition Functions

- R_0 stores the triangle id as keys and identifiers of the points forming the triangle as values. E.g $(t, (p_1, p_2, p_3))$
- R_1 stores the triangle id as keys and its neighbors with whom it shares an edge as values. E.g. $(t_1, (t_2, t_3, t_4))$
- $R_2 =$ Stores point identifier as keys and its x and y coordinates as values. E.g $(p, x \parallel y)$

Algorithm 7: Delaunay Triangulation Verification ($T = T_1, T_2, \ldots, T_n$)

```
Result: Correctness of the given Delaunay Triangulation Solution
isCorrect = True
for each $\triangle t$ in $T$ do
    $N = neighbor(t)$
    for each neighbor $\triangle n$ in $N$ do
        UNION_FIND ($\triangle n$, $\triangle t$)
        if $\triangle n$ shares an edge with $\triangle t$ and its third vertex resides outside of the circumcircle of
         $\triangle t$ then
            continue
        else
            isCorrect = False
            break
        end
    end
    if isCorrect = False then
        break
end
if All $\triangle$ are processed and belong to one set and isCorrect = True then
    print(Correct Triangulation)
else
    print(Incorrect Triangulation)
end

UNION_FIND($U$, $V$)
if $V$ does not have a parent then
    Add $U$ as $V$'s parent
else
    Update $U$'s parent to $V$'s parent
end
```

Figure 6.12

Verification algorithm of Delaunay triangulation solution

- R_3 stores the nested value in R_1 for the current triangle i.e. the OMT of neighbors of the current triangle.
- R_4 and R_5 together are used to store the disjoint sets of the triangles. R_4 stores the set information e.g. (t_2, t_1) indicate triangle t_2 belongs to the set represented by t_1. R_3 stores the member counts of the disjoint sets e.g. $(t_1, 3)$ indicates there are three triangles belonging to the set represented by triangle t_1.
- C stores the identifier of the current triangle in consideration.
- N stores the identifier of the neighbor of C in consideration.

Table 6.15

Predicates of Delaunay triangulation verification algorithm

Registers	Python Variables
R_0 : key=triangle id, value=three end points	`R0={}`
R_1 : key=triangle id, value=neighboring triangles	`R1={}`
R_2 : key=point id, value=x, y coordinates i.e. $x \parallel y$	`R2={}`
R_3 : key=neighbor triangle id	`R3={}`
R_4 : key=triangle id, value=triangle id	`R4={}`
R_5 : key=triangle id, value=member count	`R5={}`
C : current processing triangle id	`C=N=0`
N : current processing neighbor triangle id	

The verification algorithm of the Delaunay triangulation solution can be represented as five state-change functions $(F_0, F_1, F_2, F_3, F_4)$.

1. **Initialization (F_0):** This state-transition function populates R_0 with triangles with their point identifiers, R_1 with neighbor information and R_2 with coordinates of the point identifiers in

R_0. It assigns the lowest key of R_1 to C. The nested neighbor OMT of C is stored in R_3. N assigned to the lowest key of R_3 that is the neighbor of triangle in C.

Table 6.16

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0, r_1, r_2, v, C, N
Pre: $R_0 = R_1 = R_2 = R_3 = R_4 = R_5 = C = N = 0$
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $R_2 \leftarrow r_2$, $C \leftarrow get\_min(R_1)$, $(C, v) \in R_1$, $R_3 \leftarrow v$, $N \leftarrow extract\_min(R_3)$
	`def init(r0, r1, r2, v, C, N):`
	`    if not R0 and not R1 and not R2`
	`    and not R3 and not R4 and not`
	`   R5 and C==N==0:`
	`        R0=r0`
	`        R1=r1`
	`        R2=r2`
	`        C=get_min(R1)`
	`        v=R1[C]`
	`        R3=v`
	`        N=extract_min(R3)`

2. **Disjoint set initialization (F_1):** F_1 initializes R_4 and R_5. Initially each triangle is set as its own parent in R_4 and their member count is stored in R_5 (initially 1).

3. **Check circumcircle property (F_2):** This state-transition function checks two properties of neighbor N with C. First, whether it shares a side with C and second the circumcircle property i.e. ensuring that the neighbor shares an edge with the current triangle and its third endpoint does not fall within the circumcircle of the current triangle and vice-versa.

4. **Union find of triangles(F_3):** Based on the member count in R_5 of triangle sets, it assigns parent triangle in R_4. Triangle t_1 will be assigned as parent of t_2 in R_4 if the member count of t_1 is greater than the member count of t_2.

Table 6.17

Disjoint set initialization

F_1: Disjoint set initialization	F_1: Python Equivalent

Input: t
Pre: $t \notin R_4, t \notin R_5, t \in R_0, t \in R_1$
Post: $(t, t) \in R_4$, $(t, 1) \in R_5$,
	`def disjoint_set_init(t):`
	`    if R4[t]==0 and R5[t]==0 and R0[t]!=0 and R1[t]!=0:`
	`        R4[t]=t`
	`        R5[t]=1`

Table 6.18

Check circumcircle property

F_2: Check circumcircle property	F_2: Python Equivalent

Input: $p_1, p_2, p_3, p_4, x_c, y_c, r_c, x_n, y_n, r_n, v_1, v_2$
Pre: $next\_key(N, R_3) \geq N$, $\{p_1, p_2, p_3, p_4\} \in R_2$, $(C, v_1) \in R_0$, $\{p_1, p_2, p_3\} \in v_1$, $(N, v_2) \in R_0$, $\{p_2, p_3, p_4\} \in v_2$, $dist(p_1, x_c, y_c) = dist(p_2, x_c, y_c) = dist(p_3, x_c, y_c) = r_c$, $dist(p_2, x_n, y_n) = dist(p_3, x_n, y_n) = dist(p_4, x_n, y_n) = r_n$, $dist(p_1, x_n, y_n) > r_n$, $dist(p_4, x_c, y_c) > r_c$
Post: $N \notin R_3$
	`def check_circumcircle(p1, p2, p3, p4, xc, yc, rc, xn, yn, rn, v1, v2, ct):`
	` if next_key(N, R_3) >= N, R2[p1]!=0 and R2[p2]!=0 and R2[p3]!=0 and R2[p4]!=0 and R0[C]==v1 and v1[p1]!=0 and v1[p2]!=0 and v1[p3]!=0 and R0[N]==v2 and v2[p2]!=0 and v2[p3]!=0 and v2[p4]!=0 and distance(p1, xc, yc)==distance(p2, xc, yc)==distance(p3, xc, yc)==rc and distance(p2, xn, yn)==distance(p3, xn, yn)==distance(p4, xn, yn)==rn and distance(p1, xn, yn) > rn and distance(p4, xc, yc) > rc:`
	`        R3[N]=0`

Table 6.19

Union find of triangles

F_3: Union find of triangles	F_3: Python Equivalent

Input: v_1, v_2, v, ct
Pre: $N \notin R_3$, $(C, v_1) \in R_4$, $(N, v_2) \in R_4$, $v = max(v_1, v_2)$
Post: $(C, v) \in R_4$, $(N, v) \in R_4$, $(v, ct+1) \in R_5$, $N \leftarrow extract\_min(R_3)$
	`def union_find(v1, v2, v, ct):`
	`    if R3[N]==0 and R4[C}==v1 and`
	`   R4[N]==v2 and v==max(v1, v2):`
	`        R4[C]=v`
	`        R4[N]=v`
	`        R5[v]=ct+1`
	`        N=extract_min(R3)`

5. **Assign new triangle (F_4):** This state-transition function extracts the minimum key of R_1 to assign to C once all the neighbors of the previous C have been processed and checked for triangulation property. No new triangle would be assigned to C once R_1 is exhausted and R_3 will also be empty denoting the completion of the algorithm.

6.3.4 Complexity Analysis

The verification of Delaunay triangulation algorithm checks the correctness of the given triangulation of T triangles. Using conventional model, it takes $\mathcal{O}(P \log_2 P)$ time to triangulate a set of P points. In TMMC, read, write, computing minimum, maximum take logarithmic time. Each read and write in R_0, R_1, R_4 and R_5 will therefore take $\mathcal{O}(\log_2 T)$ time.

F_0 function is executed only once. It initializes few registers and performs the $get\_min()$ and $extract\_min()$ function one time.

F_1 function executes for all the triangles T in the input and makes entries in two OMTs, each in $\mathcal{O}(\log_2 T)$ time.

Table 6.20

Assign new triangle

F_4: Assign new triangle	F_4: Python Equivalent

Input: v, N, C
Pre: $R_3 = 0, next\_key(C, R_1) \geq C$
Post: $C \notin R_1$, $C \leftarrow extract\_min(R_1)$, $(C, v) \in R_1$, $R_3 \leftarrow v$, $N \leftarrow extract\_min(R_3)$
	```
	def assign_new_triangle(v, N, C):
	    if not R3 and next_key(C, R1)
	   >= C:
	        R1[C]=0
	        C=extract_min(R1)
	        if v==R1[C]:
	            R3=v
	            N=extract_min(R3)
	```

F_2 function checks the properties for each triangle T. As the number of neighbors if fixed it takes $\mathcal{O}(T \log_2 T)$ time for this check.

F_3 performs union-find for all the neighbors of each triangle T also in $\mathcal{O}(T \log_2 T)$ time. F_4 assigns a new triangle $T - 1$ time taking $\mathcal{O}(T \log_2 T)$ in total.

CHAPTER VII

NP PROBLEMS

Many natural and simple problems in computer science fall in the NP class. This class contains problems whose solution can be verified using non-deterministic polynomial (hence NP) time algorithms. NP problems are hard to solve (there are no known polynomial-time algorithms to solve them [21], but the correctness of solutions to such problems can be easily (using polynomial-time algorithms) verified. In the TMMC model, we are only interested in the verification of the correctness of solutions. Thus, towards our goal of assuring the correctness of the execution of algorithms, as our primary concern is minimizing verifier complexity, even algorithms in the NP class can be assured.

In a TMMC blockchain network, untrusted provers perform the heavy computation of solving NP problems. Along with the proof of correctness of the solution, every solution is also associated with a metric (also provided by the provers), to demonstrate how "good" the solution is. Examples of such metrics can be:

- How quick the solution is i.e. what is the time complexity?
- Does it provide an exact solution?
- Does it provide an optimal solution?

- Does it provide the best-performing approximate solution?
- How many constraints were relaxed?

These solutions can be determined by the untrusted provers using the conventional model of computation. The provers can compete amongst themselves to provide a solution with the best metric. The verifier only needs to determine the correctness of the metric computation and the verification algorithm to choose the best one.

Thus, for algorithms in NP class, a solution O can be determined using the traditional VN model of computing. For such problems, there is often a metric m_o associated with the solution O. Given a solution O with metric m_o (for an algorithm $f()$ operating on input I) the TMMC algorithm is merely intended to establish that the solution is correct and that the metric is indeed m_o. When such algorithms are executed in the TMMC blockchain, the incentivized user may compete to provide the solution with the best metric and execute the verification algorithm to prove the correctness of the solution. Therefore, TMMC allows two major advantages while considering NP problems:

1. Proving the correctness of an NP problem solution only involves verifying it in logarithmic time. It is none of the verifiers' concern on how the problem was solved.
2. In scenarios where the provers compete to provide the optimal/best solution, the verifiers only need to find the solution with the best metric and verify its correctness.

TMMC takes advantage of distributed computing in blockchain to provide a platform that dramatically simplifies the work that needs to be performed by verifiers. More specifically, the verification complexity is practically constant, viz., independent of the size/scale of inputs. In this section, we investigate TMMC verification algorithms of several common NP problems like Maximal Independent Set (MIS), Vertex Cover (VC), and the Travelling Salesman Problem (TSP), Graph Coloring problem, Clique problem, Dominating set problem, etc. [29].

7.1 Maximal Independent Set (MIS)

Given an undirected graph $G = (V, E)$, an independent set is a subset of nodes $U \subseteq V$, such that no two nodes in U are adjacent. Maximal Independent Set (MIS) is an independent set but with a special property Figure 7.1.

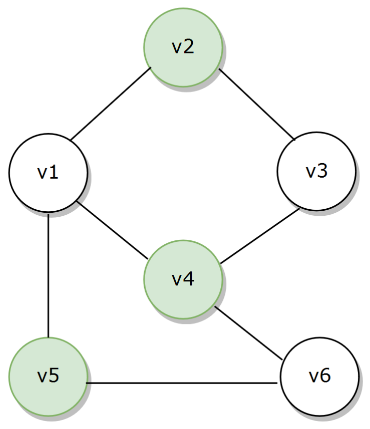


Figure 7.1

Maximal independent set $\{v_2, v_4, v_5\}$

An independent set is maximal if no node can be added without violating independence. If I is an MIS of graph G with vertices V, then no new vertex can be added to the set I without breaking the adjacent property.

Finding a maximal independent set (MIS) of a graph using a sequential algorithm is difficult. The decision version of the problem is given a graph $G = (V, E)$, does G contain a maximal independent set of cardinality k? Now, if we are given the solution S, all we have to do is prove S is the maximal independent set of G. The verification algorithm we are proposing is given in Figure 7.2:

```
Algorithm 8: MIS-Verification(G, S)
Result: $S$ is the MIS($S$)
G(V, E)
for each $s \in S$ do
    for each $n \in neighbor(s)$ do
        if $n \in S$ then
            return false
        end
        else
            $V = V - n$
        end
    end
    $V = V - s$
end
if $V = \emptyset$ then
    return true
end
else
    return false
end
```

Figure 7.2

Verification algorithm for MIS

7.1.1 Predicates and State-transition Functions

We need three OMTs to execute the MIS verification algorithm. The description of these OMTs along with some other registers to store constants are given below:

- R_0 stores the root of the OMT containing the input graph with the node as key and the nested OMT representing the neighboring nodes as value.
- R_1 stores the root of the OMT containing the MIS solution with the node as key and a dummy as a value as we do not need any other information such as edge weight in MIS solution.
- R_2 stores the root of nested OMT from R_0 containing the neighboring node as key and a dummy as a value.
- C stores the identifier of the solution node currently being processed
- C_N stores the identifier of the neighbor of C currently being processed

Table 7.1

Predicates of MIS verification algorithm

Registers	Python Variables
R_0 : (n, v), n=node in input, v=adjacent node OMT	`R0={}`
R_1 : (n, v), n=node in MIS, v=dummy value	`R1={}`
R_2 : (n, v), n=neighbor node, v=dummy value	`R2={}`
C : Node in solution currently being handled	`C=0`
C_N : Neighbor of C currently being handled	`Cn=0`

We need two state-transition functions for the MIS verification algorithm:

1. **Initialize (F_0):** The invocation of this state-change function populates two OMTs (R_0 and R_1). Every item in R_0 represents each node of the graph. Key of R_0 is a unique node identifier v and value is a nested OMT representing nodes adjacent to v (assigned to R_2. In the nested OMT, keys represent a unique identifier of the adjacent nodes. Every item in

R_1 represents each node in the MIS set i.e. the solution. The key of R_1 is a unique node identifier v'.

After the execution of this state-change function, R_0 and R_1 are populated with the input graph and the solution respectively. At the end, the lowest key of R_1 is assigned as the current node C, the lowest key of R_2 is assigned as the current node C_N for the next state-change function.

Table 7.2

Initialize

F_0: Initialization	F_0: Python Equivalent

Input: r_0, r_1, v, C, C_N
Pre: $R_0 = R_1 = R_2 = C = C_N = 0$
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $C \leftarrow get\_min(R_1)$, $(C, v) \in R_0$, $R_2 \leftarrow v$, $C_N \leftarrow get\_min(R_2)$
	`def init(r0, r1, v, C, Cn):`
	`    if not R0 and not R1 and not R2`
	`    and C==0 and Cn==0:`
	`        R0=r0`
	`        R1=r1`
	`        C=get_min(R1)`
	`        if R0[C]==v:`
	`            R2=v`
	`            Cn=get_min(R2)`

2. **Verifying neighboring property (F_1):** This state-change function verifies the neighboring property for the current node in C. This means the neighboring node of C in C_N should not be in R_1. This neighboring property preserves the definition of the maximal independent set.

   At the end of executing this state-change function, C_N is deleted from R_2 and assigned to the min key in R_2. If the next key is lower than the current key, then we can confirm the neighboring property of all the neighbors of C is completed.

Table 7.3

Verifying neighboring property

F_1: Verifying neighboring property	F_1: Python Equivalent

Input: r_0, r_1, C_N
Pre: $next\_key(C_N, R_2) \geq C_N, C_N \notin R_1$
Post: $C_N \notin R_1$, $C_N \leftarrow extract\_min(R_2)$, $C_N \notin R_0$
	`def verify_neighbor(Cn):`
	`    if next_key(Cn, R2) > Cn and Cn`
	`    not in R1:`
	`        R1[Cn]=0`
	`        Cn=extract_min(R2)`
	`        R0[Cn]=0`

3. **Update current node (F_2):** This state-change function updates the current node C to the min key in R_1. It makes sure the register R_2 is empty indicating the neighbors of the previous C were checked to satisfy the neighboring property. It also assigns R_2 to the neighbors of the new node in C.

   At the end of executing this state-change function, C is deleted from R_0. If the next key is lower than the current key, then we can confirm we have considers all the nodes in the solution in R_1. The content of R_0 and R_2 will be empty when the given solution is correct.

7.1.2 Complexity Analysis

MIS algorithm finds the maximal independent set in graph G with V nodes and E edges. In our representation, each atomic state-transition function is associated with different complexity.

The initialization function F_0 is invoked only once for initializing R_0 and R_1 with the input and the solution of the MIS problem in constant time. It takes $\mathcal{O}(\log_2 V)$ time for the $get\_min()$ function and to check existence of an item.

Table 7.4

Update current node

F_2: Update Current Node	F_2: Python Equivalent

Input: v, C
Pre: $next\_key(C, R_1) \geq C, (C, v) \in R_0, R_2 = 0$
Post: $C \notin R_0$, $C \notin R_1$, $C \leftarrow extract\_min(R_1)$, $R_2 \leftarrow v$
	`def update_current_node(v, C):`
	`    if next_key(C, R1) > C and R0[C]==v and R2==0:`
	`        R0[C]=0`
	`        R1[C]=0`
	`        C=extract_min(R1)`
	`        R2=v`

F_1 checks the neighboring property for each neighbor of the current node and is invoked for $(V-1)$ neighbors of V total nodes. Each checking takes $\mathcal{O}(\log_2 V)$ time meaning $\mathcal{O}(V^2 \log_2 V)$ time in total.

F_2 updates the current node $(V-1)$ times. Each update takes $\mathcal{O}(\log_2 V)$ time meaning $\mathcal{O}(V \log_2 V)$ in total.

7.2 Vertex Cover (VC)

Vertex Cover of a graph $G = (V, E)$ is a set of nodes, S such that each edge in E has at least one endpoint in S. This means nodes in S will cover all the edges of G. If $(u, v) \in E$, then $u \in S$ and/or $v \in S$.

Finding a vertex cover of a graph is a hard problem. The decision version of this problem is given the graph G and a positive integer k, does G have a vertex cover of size k? which is an NP problem. As we know NP problem can be verified easily (in polynomial time) once given the solution, we propose in Figure 7.4 a verification algorithm to verify the correctness of the given vertex cover.

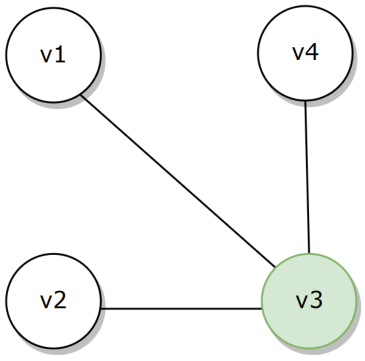


Figure 7.3

Vertex cover $\{v_3\}$

```
Algorithm 9: VC-Verification(G, C)
Result: $C$ is the Vertex Cover of $G = (V, E))$
$E' = G.E$
for each edge $e = \{u, v\}$ in $E$ do
    if $u \in C$ and/or $v \in C$ then
        $E' = E' - \{u, v\}$
    end
end
if $E' = \emptyset$ then
    return true
else
    return false
end
```

Figure 7.4

Verification algorithm for vertex cover

7.2.1 Predicates and State-transition Functions

We need two OMTs to execute the VC verification algorithm. The description of these OMTs along with some other registers to store constants and state-transition functions are given below:

- R_0 stores the root of the OMT containing the edges of the input graph. Every item in R_0 represents each edge of the graph. The key of R_0 is a unique edge identifier of the form $u \parallel v$ where u and v are node identifiers concatenated together.
- R_1 stores the provided VC solution. The key of R_1 is a unique node identifier v and the values are dummies.
- C_E holds the current edge in R_0 being processed
- U holds the start node of the current edge being processed.
- V holds the end node of the current edge being processed

Table 7.5

Predicates of vertex cover verification algorithm

Registers	Python Variables

R_0 : $(u \parallel v, d)$, $u \parallel v$=edge with node u and v, d=dummy value
R_1 : (n, d), n=node in MVC, d=dummy value
C_E : Stores the current edge being processed
U : Stores the one endpoint of the current edge
V : Stores the other endpoint of the current edge
	`R0={}`
	`R1={}`
	`Ce=0`
	`U=0`
	`V=0`

We need three state-change functions to execute the VC verification algorithm.

1. **Initialization (F_0):** After the execution of this state-change function, R_0 and R_1 are populated with the edges of the input graph and the solution respectively. At the end, the lowest key of R_0 is assigned to C_E as the current edge for the next state-change functions. Another two registers U and V are assigned to the two nodes associated with the edge in C_E.

   Every item in R_0 represents each edge of the graph. Key of R_0 is a unique node identifier $u \parallel v$. Every item in R_1 represents each node in the VC set i.e. the solution. The key of R_1 is a unique node identifier v.

Table 7.6

Initialization

F_0: Initialization	F_0: Python Equivalent
	`def init(r0, r1, Ce, U, V):`
	`    if not R0 and not R1 and Ce==U`
	`   ==V==0:`
Input: r_0, r_1, C_E, U, V	`        R0=r0`
Pre: $R_0 = R_1 = C_E = U = V = 0$	`        R1=r1`
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $C_E \leftarrow get\_min(R_0)$,	`        Ce=get_min(R0)`
$U \leftarrow split(C_E)[0]$, $V \leftarrow split(C_E)[1]$	`        endpoints=split(',')`
	`        U=endpoints[0]`
	`        V=endpoints[1]`

2. **Remove edge with start node in solution (F_1):** This state-change function checks whether the node in U is in R_1. If it is, then the edge in C_E is removed from R_0.

3. **Remove edge with end node in solution (F_2):** This state-change function checks whether the node in V is in R_1. If it is, then the edge in C_E is removed from R_0.

Table 7.7

Remove edge with start node in solution

F_1: Remove edge with start point	F_1: Python Equivalent

Input: C_E
Pre: $next\_key(C_E, R_0) \geq C_E, U \neq 0, U \in R_1$
Post: $C_E \notin R_0$, $C_E \leftarrow extract\_min(R_0)$, $U \leftarrow split(C_E)[0]$, $V \leftarrow split(C_E)[1]$
	`def remove_edge_with_start_node(Ce)`
	`    :`
	`     if next_key(Ce, R0) >= Ce and U`
	`    =!0 and R1[U]!=0:`
	`         R0[Ce]=0`
	`         Ce=extract_min(R0)`
	`         U=Ce.split(',')[0]`
	`         V=Ce.split(',')[1]`

Table 7.8

Remove edge with end node in solution

F_2: Remove edge with end point	F_2: Python Equivalent

Input: C_E
Pre: $next\_key(C_E, R_0) \geq C_E, V \neq 0, V \in R_1$
Post: $C_E \notin R_0$, $C_E \leftarrow extract\_min(R_0)$, $U \leftarrow split(C_E)[0]$, $V \leftarrow split(C_E)[1]$
	`def remove_edge_with_start_node(Ce)`
	`    :`
	`     if next_key(Ce, R0) >= Ce and V`
	`    =!0 and R1[V]!=0:`
	`         R0[Ce]=0`
	`         Ce=extract_min(R0)`
	`         U=Ce.split(',')[0]`
	`         V=Ce.split(',')[1]`

If the solution is correct all the edges will be covered by the nodes in the solution and the content of R_0 will be empty.

7.2.2 Complexity Analysis

Vertex Cover finds the vertex cover in graph G with V nodes and E edges. Using conventional architecture. In our representation, each atomic function is associated with different complexity. In the TMMC model, every read and write from memory takes logarithmic time.

Initialization function F_0 is invoked only once and assigns roots to registers in constant time. It calls the $get\_min()$ function in $\mathcal{O}(\log_2 E)$ time.

F_1 removes all the edges from R_0 that have their start node in the solution. Removing one edge takes $\mathcal{O}(\log_2 E)$ time meaning $\mathcal{O}((E/2)\log_2 E)$ in total.

F_2 removes all the edges from R_0 that have their end node in the solution. Removing one edge takes $\mathcal{O}(\log_2 E)$ time meaning $\mathcal{O}((E/2)\log_2 E)$ in total.

7.3 Traveling Salesman Problem (TSP)

Travelling Salesman Problem is closely related to the Hamiltonian-cycle problem. Hamiltonian cycle in a graph G contains a path that covers all the nodes of G i.e. V and no node in V is visited two times except the starting node (Figure 7.5). Traveling salesman is an NP-Hard Problem. The naive and dynamic solution to this problem is infeasible. The approximate solution basically solves it using Prim's algorithm for calculating a minimum spanning tree if a certain condition is satisfied.

On the other hand, given the solution to this problem, we can easily verify its correctness (in polynomial time). It refers to checking whether the given solution forms a Hamiltonian path in the

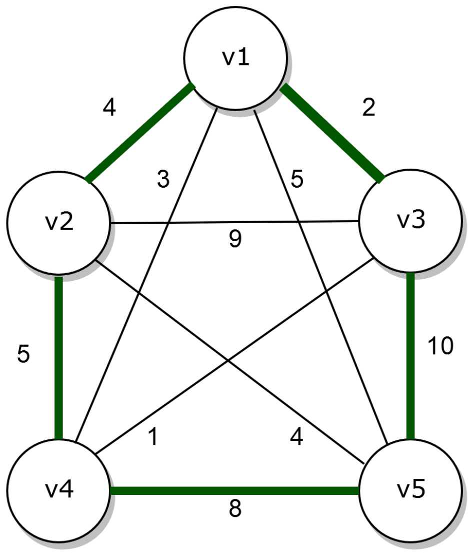


Figure 7.5

TSP solution (Hamiltonian cycle $\{v_1, v_2, v_4, v_5, v_3, v_1\}$ with start node v_1)

input graph. However, the provided solution may not be the optimal solution to solve the traveling salesman problem. The verification algorithm for the TSP solution is given in Figure 7.6:

7.3.1 Predicates and State-transition Functions

We need five registers for executing the verification algorithm of the TSP solution. Three registers to store OMTs (R_0, R_1, R_2) and two registers (C, S) to store constant values.

- R_0 stores the nodes of the graph. Key of R_0 is a unique identifier of the node e.g. v.
- R_1 stores the edges of the graph. Key of R_1 is a unique identifier for the edge e.g. $u \parallel v$.
- R_2 stores the edges of TSP solution forming a Hamiltonian cycle. Key of R_2 is a unique identifier for the edge e.g. $u \parallel v$.
- S stores the start node of the TSP solution.

```
Algorithm 10: TSP Verification $(G, E')$
Result: TSP Solution $E'$ forms a Hamiltonian path
$start\_node = current\_node =$ any node u $\in V[G]$
while $current\_node \neq \emptyset$ do
    for $each\ e \in E'$ do
        if $e = (u, v)$ and $e \in E$ then
            $current\_node = v$
        end
        if $start\_node = current\_node$ and e is the last edge in E' then
            current_node = $\emptyset$
            break
        end
    end
end
```

Figure 7.6

TSP verification algorithm

- C stores the node of the graph from R_0 currently being processed.

We need two state-transition functions for executing the verification algorithm of the TSP solution.

1. **Initialization (F_0):** Invocation of this state-change function populates three OMTs (R_0, R_1, and R_2) with the input nodes, input edges and edges in the TSP solution respectively. It also gets the minimum key of R_0 and assigns it to C and S.

2. **Delete edge if starts with current node (F_1):** F_1 finds the edge with node C in the solution i.e. R_2 and if there is an edge with C as a start node it deletes that from the solution. it also makes sure the edge belongs to the original input graph by checking its existence in R_1. The end node of the deleted edge should be unvisited, meaning should not be in R_0.

Table 7.9

Predicates of TSP solution verification algorithm

Registers	Python Variables

$R_0 : (n, d)$, n=node identifier, d=dummy value
$R_1 : (u \parallel v, d)$, $u \parallel v$=edge with node u and v, d=dummy value
$R_2 : (u \parallel v, d)$, $u \parallel v$=edge with node u and v in the TSP solution, d=dummy value
C : Stores the current node being processed
S : Stores the start node
	`R0={}`
	`R1={}`
	`R2={}`
	`C=0`
	`S=0`

Table 7.10

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0, r_1, r_2, C, S
Pre: $R_0 = R_1 = R_2 = C = S = 0$
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $R_2 \leftarrow r_2$, $C \leftarrow get\_min(R_0)$, $S \leftarrow C$
	`def init(r0, r1, r2, C, S):`
	`    if not R0 and not R1 and C==S`
	`   ==0:`
	`        R0=r0`
	`        R1=r1`
	`        R2=r2`
	`        C=extract_min(R0)`
	`        S=C`

At the end of executing this function, C is also deleted from R_0 and it is reassigned to the lowest key of R_0.

Table 7.11

Delete edge if starts with current node

F_1: Delete edge if starts with current node	F_1: Python Equivalent

Input: v, C
Pre: $next\_key(C, R_0) > C$, $C \parallel v \in R_2$, $C \parallel v \in R_1$, $v \notin R_0$
Post: $C \notin R_0$, $C \leftarrow extract\_min(R_0)$, $C \parallel v \notin R_2$
	`def delete_edge(v, C):`
	`    if next_key(C, R0) > C and R2[C+v]!=0 and R1[C+v]!=0 R0 and R0[v]==0:`
	`        R0[C]=0`
	`        C=extract_min(R0)`
	`        R2[C+v]=0`

3. **Check cycle completion (F_2):** This performs a similar task as F_1 except the end node of the edge to be deleted from R_1 and R_2 should be the start node in S to complete the Hamiltonian cycle.

   At the end of executing this function, the contents of R_0 should be empty denoting all the nodes were visited by the cyclic path mentioned in the solution i.e. R_2 which will also be empty as each edge in R_2 corresponds to one node in R_0/

7.3.2 Complexity Analysis

F_0 is invoked only one time and performs the initialization in constant time. It also performs $get\_min()$ in $\mathcal{O}(\log_2 V)$ time.

F_1 runs for each node in R_0 and for each node it finds the edge in R_2 in $\mathcal{O}(\log_2 E)$ time. Its running time for all the nodes is hence $\mathcal{O}(V \log_2 E)$ time.

Table 7.12

Check cycle completion

F_2: Check cycle completion	F_2: Python Equivalent

Input: v, C
Pre: $next\_key(C, R_0) = C$, $C \parallel v \in R_2$, $C \parallel v \in R_1$, $v = S$
Post: $C \notin R_0$, $C \leftarrow extract\_min(R_0)$, $C \parallel v \notin R_2$
	`def cycle_completion(v, C):`
	`    if next_key(C, R0) > C and R2[C+v]!=0 and R1[C+v]!=0 R0 and v==S:`
	`        R0[C]=0`
	`        C=extract_min(R0)`
	`        R2[C+v]=0`

F_2 is invoked one time where it extracts the last node in R_0 in $\mathcal{O}(\log_2 V)$ time

7.4 Graph coloring

In graph coloring problem, the elements of a graph G is assigned colors. These elements can be nodes or edges. Node coloring is the most common form of graph coloring. This refers to coloring each of the node of G in such a way that no two adjacent node of G has the same color. The minimum number color needed to color the nodes of G is also called the chromatic number of G. Finding the chromatic number of a graph is a NP-problem.

7.4.1 Predicates and State-transition Functions of Graph Coloring Verification Algorithm

- R_0 is the OMT that stores the input graph G. Each entry in this OMT has the form (k, v), where k is the node identifier and v is a nested OMT containing all the neighbors of k

- R_1 contains the OMT representing colors that are needed to color the nodes of the input graph. Each entry of R_1 represents a color. If there are n colors in R_1, this means a minimum

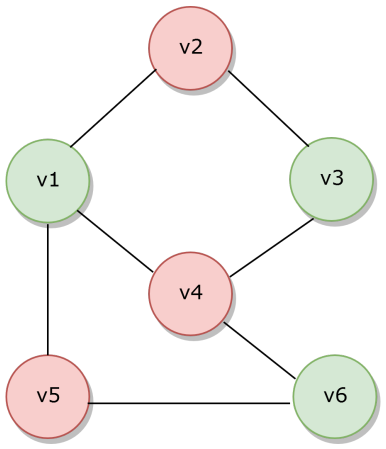


Figure 7.7

Graph colored with two colors, $\{v_1, v_3, v_6\}$=green, $\{v_2, v_4, v_5\}$=red

n color is needed to color the nodes of the input graph preserving the 'no-adjacent-same-colored-node' property.

- R_2 holds the nested value of R_0 i.e. the OMT of neighbors of the current node. Each entry in R_2 looks like (k, v) where k is the neighbor's node identifier and v is a dummy value.
- C stores the current node being handled in R_1.
- N stores the current neighbor node being handled in R_2.

Verification of the given graph coloring of a graph requires three state-transition functions using TMMC.

1. **Initialization (F_0):** F_0 initialize R_0 with the root of OMT representing the input graph. It also initializes R_1 with the root of OMT that contains all the node identifiers as keys and their corresponding colors as values. It sets C, the current node as the minimum key of R_1 and N as the minimum key of R_2 where the content of R_2 is the neighbors of C in R_0.

Table 7.13

Predicates and State-transition Functions

Registers	Python Variables

R_0 : (n, v), n=node in input, v=nested neighbor OMT
R_1 : (n, c), n=node in solution, c=color assigned
R_2 : (n, v), n=neighbor node, v=dummy value
C : Node in solution currently being handled
N : Neighbor of C currently being handled
	`R0={}`
	`R1={}`
	`R2={}`
	`C=0`
	`N=0`

Table 7.14

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0, r_1, C, N, v
Pre: $R_0 = R_1 = R_2 = C = N = 0$
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $C \leftarrow get\_min(R_1)$, $(C, v) \in R_0$, $R_2 \leftarrow v$, $N \leftarrow extract\_min(R_2)$
	`def init(r0, r1, C, N, v):`
	    `if not R0 and not R1 and not R2`
	    `and C==0 and N==0 and v==0:`
	        `R0=r0`
	        `R1=r1`
	        `C=get_min(R1)`
	        `if R0[C]==v:`
	            `R2=v`
	            `N=extract_min(R2)`

2. **Check colors of neighbor (F_1):** F_1 checks the color assignment to the current node C with the color assignment of one of its neighbors in N. Both of them should be assigned different colors.

Table 7.15

Check colors of neighbor

F_1: Check colors of neighbor	F_1: Python Equivalent
Input: c_1, c_2, N Pre: $\mathit{next\_key}(N, R_2) \geq N$, $(C, c_1) \in R_1$, $(N, c_2) \in R_1$, $c_1 \neq c_2$ Post: $N \leftarrow \mathit{extract\_min}(R_2)$	`def check_neigh_color(c1, c2, N): if next_key(N, R2) >= N and R1[C]==c1 and R1[N]==c2 and c1 != c2: N=extract_min(R2)`

3. **Assign next current node (F_2):** When we are done with checking the colors of all the neighbors in R_2, F_2 assigns a new current node to C. At the end, R_1, the solution OMT will be empty indicating all the nodes in the solution were correctly colored.

7.4.2 Complexity Analysis

Lets take graph G as our input with V nodes. F_0 is executed one time for initialization with $\mathcal{O}(1)$ time complexity. F_2 checks the colors for the given two nodes in $\mathcal{O}(\log_2 V)$ time. This function is executed for all the neighbor (at most V) nodes of all V nodes of the graph G ($\mathcal{O}(V^2 \log_2 V)$ in total).

Table 7.16

Assign next current node

F_2: Assign next current node	F_2: Python Equivalent

Input: C, v
Pre: $next\_key(C, R_1) \geq C, R_2 = 0$
Post: $C \leftarrow extract\_min(R_1)$, $(C, v) \in R_0$, $R_2 \leftarrow v$
	```
	def assign_next_node(C):
	    if next_key(C, R1) >= C and R2
	   ==0:
	        C=extract_min(R1)
	        if R0[C]==v:
	            R2=v
	```

7.5 Clique

The clique problem in computer science refers to finding cliques in graphs. A clique in a graph G with node V and edge E refers to the subset $V' \subseteq V$, where all the nodes belonging to V' are adjacent to each other. The graph containing the set of V' is also referred to as the complete sub-graph of G.

7.5.1 Predicates and State-transition Functions

- R_0 is the OMT that stores the edges of the input graph G. Each entry in this OMT has the form $(u \parallel v, d)$, where $u \parallel v$ represents an edge and d is a dummy value.
- R_1 contains the OMT representing the nodes that have cliques in graph G. If there is 4 clique in graph G, R_1 will have 4 entries.
- C stores the current clique node being handled in R_1.
- N stores the other clique node that must have an edge with C in R_0.

Verification of the given clique solution of a graph requires three state-transition functions using TMMC.

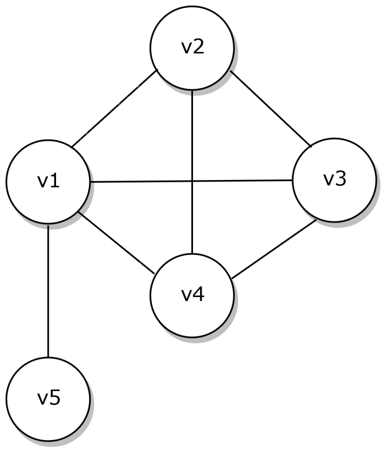


Figure 7.8

Graph with four cliques $\{v_1, v_2, v_3, v_4\}$

Table 7.17

Predicates and State-transition Functions

Registers	Python Variables

$R_0 : (u \parallel v, v)$, $u \parallel v$=edge, d=dummy value
$R_1 : (n, d)$, n=clique node, d=dummy value
C : Clique node in solution currently being handled
N : other clique node currently being checked to have an edge with C
	`R0={}`
	`R1={}`
	`C=0`
	`N=0`

1. **Initialization (F_0):** F_0 initialize R_0 with the root of OMT representing the edges of the input graph G. It also initializes R_1 with the root of OMT that contains all the nodes belongs to the clique solution. F_0 also sets C, the current clique node as the minimum key of R_1 and N which is the next key of C in R_1.

Table 7.18

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0, r_1, C_E
Pre: $R_0 = R_1 = C_E = 0$
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $C \leftarrow extract\_min(R_1)$, $N \leftarrow next\_key(C, R_1)$
	`def init(r0, r1, Ce):`
	`    if not R0 and not R1 and C==0`
	`   and N==0:`
	`        R0=r0`
	`        R1=r1`
	`        C=extract_min(R1)`
	`        N=next_key(C, R1)`

2. **Check clique edge (F_1):** F_1 checks whether there is and edge between C and N in the OMT R_0. C and N both being a clique node in the solution in R_1, there must be an edge between them.

3. **Choose the next clique node (F_2):** When we are done with checking the clique properties of the clique node in C, we remove it from R_1 and choose another node in R_1 to check the clique properties. At the end, R_1, the solution clique OMT will be empty indicating all the nodes in the solution are adjacent to each other.

Table 7.19

Check clique edge

F_1: Check clique edge	F_1: Python Equivalent

Input: C, N
Pre: $C \parallel N \in R_0, next\_key(N, R_1) \geq N$
Post: $N \leftarrow next\_key(N, R_1)$
	`def check_clique_edge(C, N):`
	`    if R0[C+N] != 0 and next_key(N,`
	`    R1) >= N:`
	`        N=next_key(N, R1)`

Table 7.20

Choose next clique node

F_2: Choose next clique node	F_2: Python Equivalent

Input: C
Pre: $next\_key(C, R_1) \geq C$
Post: $C \leftarrow extract\_min(R_1)$
	`def choose_next_clique_node(C):`
	`    if next_key(C, R1) >= C:`
	`        C=extract_min(R1)`

7.5.2 Complexity Analysis

Lets take graph G as our input with V nodes and E edges. F_0 is executed one time for initialization with $\mathcal{O}(1)$ time complexity. F_2 checks for k edges in R_0 (for k clique nodes) in $\mathcal{O}(k^2 \log_2 V)$ time. F_2 choose the next clique node $k-1$ times, each taking $\mathcal{O}(\log_2 K)$ time.

7.6 Dominating Set

In graph theory, dominating set D of a graph G with nodes V and edge E contains a subset of nodes ($D \subseteq V$) with a special property. The property is all of the node in V is adjacent to at least one node in D. The term dominating number refers to the smallest number of nodes consisting the dominating set. Finding the dominating set of a graph is a NP problem.

In Figure 7.9, the subset of nodes i.e. $\{v_1, v_5, v_6\}$ is a dominating set as every other node in V i.e. $\{v_2, v_3, v_4\}$ is adjacent to at least one node in D.

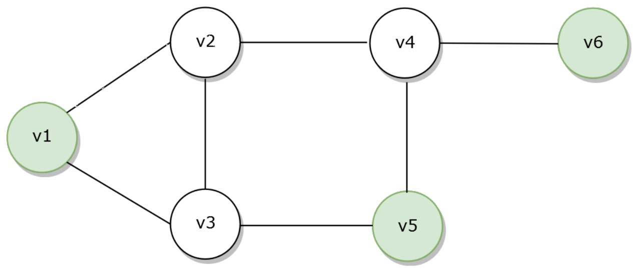


Figure 7.9

Dominating set $\{v_1, v_5, v_6\}$

7.6.1 Predicates and State-transition Functions

- R_0 is the OMT that stores the input graph G. Each entry in this OMT has the form (k, d), where k is the node identifier and d is a dummy value
- R_1 is the OMT that stores the edges of the input graph G. Each entry in this OMT has the form $(u \parallel v, d)$, where $u \parallel v$ is the edge identifier and d is a dummy value
- R_2 holds the given dominating set solution that needs to be verified. Each entry in R_2 looks like (k, d) where k is a node identifier in the dominating set and d is a dummy value.
- C stores the current node being handled in R_0.

Verification of the given dominating set of a graph requires two state-transition functions using TMMC.

Table 7.21

Predicates and State-transition Functions

Registers	Python Variables

R_0 : (n, d), n=node in input, d=dummy value
R_1 : $(u \parallel v, d)$, $u \parallel v$=edge with node u and v, d=dummy value
R_2 : (n, d), n=node in dominating set, d=dummy value
C : Node in dominating solution currently being handled
	`R0={}`
	`R1={}`
	`R2={}`
	`C=0`

1. **Initialization (F_0):** F_0 initialize R_0 with the root of OMT representing the input graph nodes and R_1 with the root of OMT that contains the edges of the input graph. It also initializes R_2 with the dominating set OMT root. It sets C, the current node as the minimum key of R_0.

2. **Check dominating set property of the current node (F_1):** F_1 checks the dominating set property of the node stored in C. This means, the current node should not belong to the dominating set and there must be at one node in R_2 with which C will have an edge in R_1. F_1 checks this property for all nodes in R_0.

   After all checking are completed, R_0 should be empty indicating the given solution to be correct.

7.6.2 Complexity Analysis

Lets take graph G as our input with V nodes and E edges. F_0 is executed one time for initialization of R_0, R_1, and R_2 with $\mathcal{O}(1)$ time complexity. F_2 checks the dominating property by reading items from R_1 and R_2 in logarithmic time i.e. $\mathcal{O}(\log_2 E)$ and $\mathcal{O}(\log_2 V)$ for V nodes ($\mathcal{O}(V(\log_2 E + \log_2 V))$in total).

Table 7.22

Initialization

F_0: Initialization	F_0: Python Equivalent

Input: r_0, r_1, r_2, C
Pre: $R_0 = R_1 = R_2 = C = 0$
Post: $R_0 \leftarrow r_0$, $R_1 \leftarrow r_1$, $R_2 \leftarrow r_2$, $C \leftarrow extract\_min(R_0)$
	`def init(r0, r1, C):`
	`    if not R0 and not R1 and not R2`
	`    and C==0:`
	`        R0=r0`
	`        R1=r1`
	`        R2=r2`
	`        C=extract_min(R0)`

Table 7.23

Check dominating set property of the current node

F_1: Check dominating set property of the current node	F_1: Python Equivalent

Input: C, n
Pre: $next\_key(C, R_0) \geq C$, $C \in R_0$, $C \notin R_2$, $n \in R_2, C \parallel n \in R_1$
Post: $C \leftarrow extract\_min(R_0)$
	`def check_domin_property(C, n):`
	`    if next_key(C, R20) >= C and R0`
	`    [C]!=0 and R2[C]==0 and R2[n]!=0`
	`    and R1[C+n] != 0:`
	`        C=extract_min(R0)`

CHAPTER VIII

CONCLUSION

Security is one of the most prime issues these days considering the amount of data-driven systems that are used in our day to day lives. Information systems are comprised of complex processes. The more complex a system, the more difficult it is to assure its integrity. Adding many layers of “security solutions” often further complicates the system, and consequently, may even increase the risk of security failures. The integrity of an information system is predicated on the integrity of processes that manipulate data and depend on the correct execution of its processes.

TCB minimizing model of computation (TMMC) explicitly seeks to minimize the TCB of a computing system that is susceptible to illegitimate attacks. Notwithstanding limitations imposed on the TCB, even complex large-scale processes can be executed under the TMMC model. It can be an alternative for the assured execution of a variety of algorithms used in practical systems.

This ongoing research work is spreading its branches to explore an increasing number of algorithms used in popular application domains. Exploring different domains helps to create a generic framework of writing state-transition functions for any algorithms. Every algorithm is different considering how its data items are stored and its processes are executed.

The boom of the blockchain network has shown mankind a groundbreaking way of securing existing systems along with some already existing cryptography tools. The application of the blockchain network is limitless and the area is very promising. This research work puts together ideas to utilize this field of enormous potential.

8.1 Contribution

With the existing active approach to deal with security attacks, we need to be ready for any eventuality. The type of attacks and threats are unlimited. Everyday, the attackers will invent a new way to create disruption. This continued battle will go on forever. Taking a passive approach where we strictly define a set of rules for executing a system will proactively make sure the integrity of the system. Any deviation from these rules will immediately trigger anomaly and we will exactly "what" went wrong and "when" it went wrong without extensive deep dive into the system. TMMC plays an important role towards moving to such definition of process execution.

As the outcomes of this research work, we have published the following papers so far:

- Naila Bushra, Naresh Adhikari and Mahalingam Ramkumar (2018, September). A TCB Minimizing Model of Computation. In International Symposium on Security in Computing and Communication (pp. 455-470). Springer, Singapore. [12]
- Naresh Adhikari, Naila Bushra and Mahalingam Ramkumar (2017). Secure Queryable Dynamic Maps. In Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government (EEE) (pp. 65-71). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). [1]

Following is a list of papers that were accepted in several conference proceeding so far.

- Naila Bushra, Naresh Adhikari and Mahalingam Ramkumar, Secure Queryable Dynamic Graphs Using Blockchains, in proceedings of IEEE TPS 2019, Los Angeles, CA
- Naila Bushra, Naresh Adhikari and Mahalingam Ramkumar, Scalable Verification of NP Problems in Blockchain, in proceedings of ICCWS 2020, Norfolk, VA
- Naresh Adhikari, Naila Bushra, and Mahalingam Ramkumar, Complete Merkle Hash Trees for Large Dynamic Spatial Data, International Conference on Computational Science and Computational Intelligence, in proceedings of CSCI 2019, Las Vegas, NV
- Naresh Adhikari, Naila Bushra, and Mahalingam Ramkumar, Blockchain-based Redistricting Evaluation Protocol, in proceedings of IEEE TPS 2019, Los Angeles, CA
- Naresh Adhikari, Naila Bushra and Mahalingam Ramkumar, Blockchain-based Redistricting Evaluation Protocol, in proceedings of ICCWS 2020, Norfolk, VA

We have successfully submitted two more manuscripts to the ACM CODASPY conference:

- Naila Bushra, Naresh Adhikari, and Mahalingam Ramkumar, Assured Execution of Computational Geometry Algorithms, CODASPY 2020, New Orleans, LA
- Naresh Adhikari, Naila Bushra, and Mahalingam Ramkumar, Authoritative and Unbiased Responses to Geographic Queries, CODASPY 2020, New Orleans, LA

We are in the process of submitting two journal papers from this research work:

- Design and Application of Secure Queryable Dynamic Graph
- TMMC: An Adaptable Model of Computation with Minimal TCB

8.2 Application of TMMC

8.2.1 Graph algorithms

The graph algorithms are essential components of many large-scale information systems. Guaranteeing the integrity of the execution of various graph algorithms (such as computation of shortest path, disjoint sets, minimum spanning tree, etc.) becomes especially challenging for graphs with very large numbers of nodes and/or edges. TMMC can verify the correctness of the execution of graph algorithms, irrespective of the scale of the graph, without the need to rely on trusted third parties. A possible application can be secure queryable dynamic graphs.

8.2.2 Computational geometry algorithms

Assured execution of computational geometry algorithms is a crucial requirement for several real-world applications in geographical information systems (GIS), computer graphics and computer-aided design, robotics, etc. A variety of computational geometry algorithms can be executed under the TMMC model.

8.2.3 Financial systems

Another area critical to secure execution is algorithms in finance. Financial institutions such as banks, credit unions, stock-markets use a range of algorithms to determine the credibility of clients, trading, fund re-balancing, etc. These algorithms are generally very complex to understand hence can be vulnerable to unauthorized alteration when executed in an untrusted environment. TMMC can provide a platform to easily break down these complex processes into simple functions that are easier to verify.

8.2.4 Dynamic algorithms

Many information systems run in real-time and deal with ever-changing data. Here, the integrity of constantly changing data items and the effect of this change in the systems need to be verified correctly in real-time to make sure the integrity of the system is intact. TMMC offers to provide strictly defined rules that drive how the system is supposed to change and provide simple ways to verify those rules upholding the correctness of the system.

8.2.5 NP problems

The algorithm for verification of the correctness of a solution is substantially easier than the algorithm for obtaining the solution. Blockchain networks incentive mechanism can be used for proposing the best solution to an NP problem. A metric can be associated with the solution for an NP problem. In blockchain-TMMC incentivized used may compete to provide the solution with the best metric, and execute the verification algorithm to prove the correctness of the solution. Given the correct solution and the best metric, TMMC can establish the correctness of the verification algorithm.

8.3 Future Work

There are some challenges still to be considered in this work. The most challenging aspect of the research would be to describe a secure model of computation that can be generalized for all information domains. Existing state-model based approaches focus mostly on specific domains of application. They do not provide any generic framework that suits the need for executing algorithms for a variety of domains. The proposed idea intends to define a model that can be used to represent a various range of algorithms each with domain-specific authenticated data structures.

A major part of this research involves converting different algorithms into state-transition functions to fit the description of TMMC and figure out given the description of the model what is the best way to execute different algorithms. The rules need to be carefully defined as a form of byte-code in order to develop a formal language to represent state-transition rules for a range of algorithms belonging to different domains.

REFERENCES

[1] N. Adhikari, N. Bushra, and M. Ramkumar, "Secure Queryable Dynamic Maps," *Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government (EEE)*. The Steering Committee of The World Congress in Computer Science, Computer , 2017, pp. 65–71.

[2] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent authenticated dictionaries and their applications," *International Conference on Information Security*. Springer, 2001, pp. 379–393.

[3] D. Andersson, "LEOCoin,", 2016.

[4] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz, "Memory-constrained algorithms for simple polygons," *Computational Geometry*, vol. 46, no. 8, 2013, pp. 959–969.

[5] T. Asano, W. Mulzer, G. Rote, and Y. Wang, "Constant-work-space algorithms for geometric problems," *Journal of Computational Geometry*, vol. 2, no. 1, 2011, pp. 46–68.

[6] T. L. Association, "An Introduction to Libra,".

[7] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture.," *USENIX Security Symposium*, 2014, pp. 781–796.

[8] J. L. Bentley and T. A. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Transactions on computers*, , no. 9, 1979, pp. 643–647.

[9] S. Betgé-Brezetz, A. Bousquet, J. Briffaut, E. Caron, L. Clevy, M.-P. Dupont, G.-B. Kamga, J.-M. Lambert, A. Lefray, B. Marquet, et al., "Seeding the cloud: An innovative approach to grow trust in cloud based infrastructures," *The Future Internet Assembly*. Springer, 2013, pp. 153–158.

[10] A. Bowyer, "Computing dirichlet tessellations," *The computer journal*, vol. 24, no. 2, 1981, pp. 162–166.

[11] J. Brown and T. F. Knight Jr, "A minimal trusted computing base for dynamically ensuring secure information flow," *Project Aries TM-015 (November 2001)*, vol. 37, 2001.

[12] N. Bushra, N. Adhikari, and M. Ramkumar, "A TCB Minimizing Model of Computation," *International Symposium on Security in Computing and Communication*. Springer, 2018, pp. 455–470.

[13] V. Butarin, "Bootstrapping a Decentralized Autonomous Corporation: Part I," *Bit-coin Magazine*, 2013.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT press, 2009.

[15] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, and M. Overmars, "Computational geometry: Algorithms and applications, 2000," *New York, New York*, 2010.

[16] B. Delaunay, "On the empty sphere," *Bulletin of Academy of Sciences of the USSR*, 1934, pp. 7–793.

[17] G. D. Dennis, *TSAFE: Building a trusted computing base for air traffic control software*, doctoral dissertation, Massachusetts Institute of Technology, 2003.

[18] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic third-party data publication," *Data and Application Security*, Springer, 2002, pp. 101–112.

[19] J. S. Fenton, "Memoryless subsystems," *The Computer Journal*, vol. 17, no. 2, 1974, pp. 143–147.

[20] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, no. 1-4, 1987, p. 153.

[21] M. R. Garey, "Computers and intractability: A guide to the theory of np-completeness," *Revista Da Escola De Enfermagem Da USP*, vol. 44, no. 2, 1979, p. 340.

[22] I. Gat and H. J. Saal, "Memoryless execution: a programmer's viewpoint," *Software: Practice and Experience*, vol. 6, no. 4, 1976, pp. 463–471.

[23] M. D. Godfrey, "Introduction to The First Draft Report on the EDVAC by John von Neumann," .

[24] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen, "Authenticated data structures for graph and geometric searching," *Cryptographers Track at the RSA Conference*. Springer, 2003, pp. 295–313.

[25] L. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi," *ACM transactions on graphics (TOG)*, vol. 4, no. 2, 1985, pp. 74–123.

[26] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter, "The Nizza secure-system architecture," *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*. IEEE, 2005, pp. 10–pp.

[27] B. Hawkins, “Case Study: The Home Depot Data Breach,” *Retrieved January*, vol. 19, 2015, p. 2016.

[28] J. Hendricks and L. Van Doorn, “Secure bootstrap is not enough: Shoring up the trusted computing base,” *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM, 2004, p. 11.

[29] R. M. Karp, “Reducibility among combinatorial problems,” *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[30] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre Attacks: Exploiting Speculative Execution,” *ArXiv e-prints*, Jan. 2018.

[31] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, “Authentication in distributed systems: Theory and practice,” *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, 1992, pp. 265–310.

[32] B. W. Lampson, “A note on the confinement problem,” *Communications of the ACM*, vol. 16, no. 10, 1973, pp. 613–615.

[33] D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities,” *10th USENIX Security Symposium*, 2001.

[34] D. C. Latham, “Department of defense trusted computer system evaluation criteria,” *Department of Defense*, 1986.

[35] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown,” *ArXiv e-prints*, Jan. 2018.

[36] E. Lipper, B. Melamed, R. Morris, and P. Zave, “A multi-level secure message switch with minimal TCB: Architectural outline and security analysis,” *Aerospace Computer Security Applications Conference, 1988., Fourth*. IEEE, 1988, pp. 242–249.

[37] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, “Algebraic methods for interactive proof systems,” *Journal of the ACM (JACM)*, vol. 39, no. 4, 1992, pp. 859–868.

[38] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, “A general model for authenticated data structures,” *Algorithmica*, vol. 39, no. 1, 2004, pp. 21–41.

[39] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, “TrustVisor: Efficient TCB reduction and attestation,” *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

[40] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri, “How low can you go?: recommendations for hardware-supported minimal TCB code execution,” *ACM SIGARCH Computer Architecture News*. ACM, 2008, vol. 36, pp. 14–25.

[41] R. C. Merkle, “Protocols for public key cryptosystems,” *Security and Privacy, 1980 IEEE Symposium on*. IEEE, 1980, pp. 122–122.

[42] I. Miers, C. Garman, M. Green, and A. D. Rubin, “Zerocoin: Anonymous distributed e-cash from bitcoin,” *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 397–411.

[43] A. Miller, M. Hicks, J. Katz, and E. Shi, “Authenticated data structures, generically,” *ACM SIGPLAN Notices*, vol. 49, no. 1, 2014, pp. 411–423.

[44] S. Mohanty, M. Ramkumar, and N. Adhikari, “OMT: A Dynamic Authenticated Data Structure for Security Kernels,” *International Journal of Computer Networks & Communications*, vol. 8, 2016, pp. 1–23.

[45] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[46] C. Papamanthou and R. Tamassia, “Time and space efficient algorithms for two-party authenticated data structures,” *International conference on information and communications security*. Springer, 2007, pp. 1–15.

[47] B. Parno, J. Howell, C. Gentry, and M. Raykova, “Pinocchio: Nearly practical verifiable computation,” *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 238–252.

[48] D. Patten, “The evolution to fileless malware,” *Retrieved from*, 2017.

[49] M. Peinado and W. Cui, “Malware investigation by analyzing computer memory,”, Oct. 22 2013, US Patent 8,566,944.

[50] C. Rackoff and D. R. Simon, “Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack,” *Annual International Cryptology Conference*. Springer, 1991, pp. 433–444.

[51] T. Radichel, “Case study: critical controls that could have prevented target breach,” *SANS Institute InfoSec Reading Room*, 2014.

[52] M. Ramkumar, “Minimal TCB for System-Model Execution,” .

[53] M. Ramkumar and S. D. Mohanty, “Reliable Assurance Protocols for Information Systems,” *International Conference on the Evolving Internet*, 2015.

[54] J. Rushby et al., “A trusted computing base for embedded systems,” *Proceedings 7th DoD/NBS Computer Security Conference*. Citeseer, 1984, pp. 294–311.

[55] J. M. Rushby, *Design and verification of secure systems*, vol. 15, ACM, 1981.

[56] H. Saini, Y. S. Rao, and T. C. Panda, “Cyber-crimes and their impacts: A review,” *International Journal of Engineering Research and Applications*, vol. 2, no. 2, 2012, pp. 202–209.

[57] D. Samyde, S. Skorobogatov, R. Anderson, and J.-J. Quisquater, “On a new way to read data from memory,” *Security in Storage Workshop, 2002. Proceedings. First International IEEE*. IEEE, 2002, pp. 65–69.

[58] M. I. Shamos and D. Hoey, “Closest-point problems,” *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. IEEE, 1975, pp. 151–162.

[59] M. I. Shamos and D. Hoey, “Geometric intersection problems,” *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE, 1976, pp. 208–215.

[60] T. SPERRY, “386 VS 030-THE CROWDED FAST LANE,” *DR DOBBS JOURNAL*, vol. 13, no. 1, 1988, p. 16.

[61] T. SPERRY, “386 VS 030-THE CROWDED FAST LANE,” *DR DOBBS JOURNAL*, vol. 13, no. 1, 1988, p. 16.

[62] P. Stewin and I. Bystrov, “Understanding DMA malware,” *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 21–41.

[63] R. Tamassia, “Authenticated data structures,” *European Symposium on Algorithms*. Springer, 2003, pp. 2–5.

[64] V. Thotakura and M. Ramkumar, “Minimal trusted computing base for MANET nodes,” *Wireless and Mobile Computing, Networking and Communications (WiMob), 2010 IEEE 6th International Conference on*. IEEE, 2010, pp. 91–99.

[65] N. Van Saberhagen, “CryptoNote v 2.0,”, 2013.

[66] G. Voronoi, “Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs.,” *Journal für die reine und angewandte Mathematik*, vol. 134, 1908, pp. 198–287.

[67] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” *ACM SIGARCH Computer Architecture News*. ACM, 2007, vol. 35, pp. 494–505.

[68] T. Y. Woo and S. S. Lam, “Authentication for distributed systems,” *Computer*, vol. 25, no. 1, 1992, pp. 39–52.

[69] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, 2014, pp. 1–32.