Abstract

Efficient Fault-Tolerant Infrastructure for Cloud Computing

Xueyuan Su

2014

Cloud computing is playing a vital role for processing big data. The infrastructure is built on top of large-scale clusters of commodity machines. It is very challenging to properly manage the hardware resources in order to utilize them effectively and to cope with the inevitable failures that will arise with such a large collection of hardware. In this thesis, task assignment and checkpoint placement for cloud computing infrastructure are studied.

As data locality is critical in determining the cost of running a task on a specific machine, how tasks are assigned to machines has a big impact on job completion time. An idealized abstract model is presented for a popular cloud computing platform called Hadoop. Although Hadoop task assignment (HTA) is $\mathcal{NP}$-hard, an algorithm is presented with only an additive approximation gap. Connection is established between the HTA problem and the minimum makespan scheduling problem under the restricted assignment model. A new competitive ratio bound for the online GREEDY algorithm is obtained.

Checkpoints allow recovery of long-running jobs from failures. Checkpoints themselves might fail. The effect of checkpoint failures on job completion time is investigated. The sum of task success probability and checkpoint reliability greatly affects job completion time. When possible checkpoint placements are constrained, retaining only the most recent $\Omega(\log n)$ possible checkpoints has at most a constant factor penalty. When task failures follow the Poisson distribution, two symmetries for non-equidistant placements are proved and a first order approximation to optimum placement interval is generalized.

Efficient Fault-Tolerant Infrastructure for Cloud Computing

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Xueyuan Su

Dissertation Director: Michael J. Fischer

December 2013

to my parents and wife

Acknowledgments

I have been very fortunate during my stay at Yale, and this is due in no small part to my great advisor, Michael J. Fischer. I started working in theoretical computer science during my first summer here. The progress seemed hopeless at the beginning. Mike has taught me how to think like a theoretician, how to break complicated problems down into smaller pieces, solve them with persistent efforts, and connect the dots precisely in the end. I am always inspired by the advice and guidance that Mike gives to me. More importantly, I have learned the dedication and optimism that Mike shows when he approaches everything he does, not only in research, but also for the rest of life. These qualities would be invaluable to my career and life ahead.

I have benefited a lot from the interactions with my committee members. Dana Angluin has shown me humility, patience, and how to tackle research problems in general. James Aspnes has impressed me for his enthusiasm in teaching and research since I took the first course in distributed computing theory from him. A significant part of my research work has been inspired by Michael Mitzenmacher and his work on randomized load balancing. The discussions with Avi Silberschatz has greatly shaped my understanding of database systems and the cloud computing paradigm.

Many other faculty members have also contributed to my development at Yale. These include Daniel Abadi, Dirk Bergemann, Eric Denardo, Joan Feigenbaum, Bryan Ford, Gil Kalai, Drew V. McDermott, Vladimir Rokhlin, Holly Rushmeier, Zhong Shao, Daniel A.

Spielman, and Yang Richard Yang. Three of these people deserve special credit: Daniel Abadi for discussions on MapReduce and Hadoop, Bryan Ford for bringing the checkpoint retention problem into my attention, and Eric Denardo for his marvelous introduction to various optimization techniques. For several years, I had worked with Eric on his optimization course and enjoyed his interesting stories on operations research in early days.

My thanks also go to my friends who made my stay enjoyable. I thank Azza Abouzeid, Rich Alimi, Amittai Aviram, Kamil Bajda-Pawlikowski, Zhuo Han, Jiewen Huang, Liang Liang, Edo Liberty, Hongqiang Liu, Jianye Lu, John Maheswaran, Michael F. Nowlan, Patrick Paczkowski, Donya Quick, Lev Reyzin, Liangchen Song, Garret Swart, Andreas Voellmy, Bing Wang, Hao Wang, Huan Wang, Ye Wang, David Wolinsky, Hongzhi Wu, Chen Xu, Yun Xu, Su Xue, Yitong Yin, and Ennan Zhai. I especially thank Yitong Yin for great collaborations on research, and thank Garret Swart for mentoring my practice on integrating Hadoop into database systems.

I also want to acknowledge my sources of funding. My graduate studies have been funded by a Yale University Fellowship, a Frederick W. Vanderbilt Fellowship, a Sheffield Scholarship, and a Kempner Fellowship.

Last but most importantly, I want to thank my family who gave so generously of their time, care, and love. I would not have been able to finish my study without my wife Lu's constant support. Being the father of my own kid, I finally understand how challenging it is to raise a child and foster his pursuit for dreams. I am so grateful for everything that my parents have given to me throughout the years.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Cloud Computing

Cloud computing has received significant attention from both academic and industrial communities in recent years. It envisions shifting data storage and computing power away from local servers, across the network cloud, and into large data centers. Now it is playing a vital role for processing huge volumes of data in many areas.

On the one hand, cloud computing refers to the applications hosted by data centers and delivered over the Internet. Search from Google and Microsoft, video streaming from YouTube and Hulu, purchases from Amazon and eBay, and social networking from Facebook and Twitter are a few well-known examples.

On the other hand, cloud computing also refers to the hardware and software infrastructure that support those cloud applications. Cloud computing infrastructure is built on top of large-scale clusters of commodity machines. It provides good scalability and a highly parallel execution environment for applications.

1.2 MapReduce and Hadoop

Google's MapReduce [22, 23, 54] parallel computing architecture splits workload over large clusters of commodity machines and enables automatic parallelization. By exploiting parallel processing, it provides a software platform that lets one easily write and run applications that process large amounts of data. Apache Hadoop [10] is a free Java implementation of MapReduce in the open source software community. It has become the most popular cloud computing platform for many large data processing applications.

The Hadoop/MapReduce framework provides applications with a highly parallel execution environment. It consists of a data storage module and a job execution module. A Hadoop system relies on a distributed file system called the Hadoop Distributed File System (HDFS) for data storage. Large files are broken into smaller pieces called data blocks that are replicated and locally stored on hard disks of machines in the cluster. The job execution module partitions a single job into many small tasks. Tasks are executed on the same set of machines that are used for data storage. For efficiency, MapReduce tries to assign tasks to machines possessing the data to be processed.

1.3 Efficiency

Job completion time in a highly parallel execution environment can be affected by many factors.

Assume a job consists of many independent tasks. Tasks assigned to the same machine are processed sequentially. The sum of processing times of all tasks assigned to a single machine is called its load. Tasks assigned to different machines run in parallel. The job completes only when all tasks complete. Therefore, the job completion time is determined by the maximum load over all machines. Load balancing has a great impact on efficiency.

The processing time of a task may depend on which machine it is assigned to. For example, in MapReduce jobs, the time to load data often dominates the actual computation time. A task assigned to a machine not possessing the needed input data must obtain it over the network from a remote machine. This can considerably increase the time to complete the task. Moreover, the more data needs to be remotely loaded, the more congested the network becomes, which further slows the arrival of the remote data.

1.4 Fault Tolerance

Hardware failures are to be expected in any infrastructure consisting of thousands of components. The chance of job completion without any failures becomes exponentially small with increasing job processing time. Assume an execution of a job fails. After the faulty components are fixed or replaced, the lost data is recomputed by rerunning as much of the failed job as necessary. The time spent in restart and recomputation may be substantial.

A widely used technique to reduce such overhead is checkpointing. Checkpoints of the job state are taken from time to time. When a failure is detected, the system performs a rollback operation that restores the job state to one previously stored in a checkpoint, and recomputation begins from there. This avoids the need for an interrupted job always to be restarted from the beginning.

In the literature, checkpoints are assumed to be stored in totally reliable storage and can always be accessed once they have been successfully created. This assumption is never strictly true in practice. However checkpoints are stored, there is some possibility that they will become corrupted and therefore not available when needed. The issue is whether or not the non-zero checkpoint failure probability is large enough to be significant.

Replication is one approach to control the reliability of stored checkpoints. By increasing the number of replicas, the probability of them all simultaneously failing is generally reduced, but doing so takes more processing time and thus delays the completion of the job. The only justification for incurring such cost is that the checkpoint, if needed, can save the possibly much larger cost of recomputing the data it contains. Faster job completion can sometimes be achieved by using less reliable but cheaper checkpoints.

Both replication time and checkpoint reliability are affected by many factors, such as what replication schemes are used, what the replication factor is, and how replicas are distributed throughout the clustered machines. Finding an optimal checkpointing scheme is a non-trivial task.

1.5 Our Contributions

In this thesis, we investigate task assignment and checkpoint placement problems in cloud computing infrastructure.

In Chapter 2, task assignment problems in Hadoop are studied. When tasks are processed in parallel over several machines, one is interested in balancing load so as to minimize the job completion time. This is called the *minimum makespan scheduling* problem, where makespan is the job completion time. It is a central problem in scheduling theory. The general minimum makespan scheduling problem is $\mathcal{NP}$-hard [34].

The Hadoop task assignment (HTA) problem differs from the classic minimum makespan scheduling problem in that the cost of each task depends on the entire assignment. If a task is assigned to a machine that possesses a replica of the needed input data, it becomes a *local task* and incurs a *local cost*; otherwise, it is a *remote task* and incurs a *remote cost* that reflects the additional expense of remotely fetching data across the network. Moreover, the remote cost for a task my grow with increasing numbers of remote tasks due to network congestion. The resulting HTA problem is $\mathcal{NP}$-hard, even in the restricted case when all machines have equal capacity of 3 units, the remote cost function is constant, and each data block has at most 2 replicas.

A simple round robin algorithm for a variant of the HTA problem is analyzed and several interesting observations are made. One theorem reveals that the intuition is wrong that increasing the number of replicas always helps load balancing. Using round robin task assignment, adding more replicas into the system can sometimes result in worse job completion time. Another theorem shows there is a multiplicative gap in makespan between the optimal assignment and the assignment computed by round robin in the worst case. Another algorithm, FLOWT, that employs maximum flow and increasing threshold techniques, is presented for the general HTA problem. It is shown that assignments computed by FLOWT are optimal to within an additive constant that depends only on the number of machines and the remote cost function. A matching lower bound for FLOWT is also shown on a special class of problem instances.

In Chapter 3, the connection is established between the HTA problem and a variant of the minimum makespan scheduling problem in which each task may only be assigned to a specified subset of feasible servers [6]. An online version of the HTA problem is considered where tasks arrive in sequence, need to be assigned upon their arrivals, and cannot be reassigned at a later time. A simple algorithm, GREEDY, works by assigning each task to a machine that is able to process it and currently has the minimum load. Ties are broken arbitrarily. An optimal offline algorithm, OPT, knows the task sequence in advance and has unbounded computational power. The makespan achieved by GREEDY is compared to that of OPT. The maximum ratio between the makespans of GREEDY and OPT under all possible inputs is the *competitive ratio*. A new upper bound on the competitive ratio is obtained by analyzing two novel structures called *witness graphs* and *token production systems (TPSs)*, both of which have some independent interest. This bound refines the original result in [6] and is incomparable to the best such result known to date [12].

In Chapter 4, the effect of possible checkpoint failures on job completion time is studied. The reliability of a checkpoint is the probability with which each attempted access to this checkpoint succeeds. A two-level Markov chain is used to represent job execution in the presence of task and checkpoint failures. Random walks that correspond to job executions are analyzed. A general framework for computing the expected job completion time is developed, providing tools to system designers for evaluating different execution plans.

Consider the uniform case with identical tasks and checkpoints. Fix the length of each task and let the number of tasks grow. The sum of the task success probability p and the checkpoint reliability q is shown to be an important design parameter. Let n be the number of tasks. The expected job completion time grows linearly in n when $p + q > 1$, quadratically when $p+q = 1$, and exponentially when $p+q < 1$. It is also shown that when $p + q > 1$, by retaining only the most recent logarithmic number of possible checkpoints, the penalty on the expected job completion time is only a constant factor.

Non-uniformly placed checkpoints split a job into tasks of varying lengths. The vector of resulting task lengths is called the job's length vector. Assume task failures follow the Poisson distribution. Two counter-intuitive symmetry properties are proved. The first property states that reversing a length vector does not change the expected job completion time. The second property indicates that the expected completion time can never increase when the length vector is replaced by the average of itself and its reversal. A conjecture is that, when $0 < q < 1$, it is optimal to cluster checkpoints towards the middle of the job. This conjecture is proved for the special case with two checkpoints.

Let M be the mean time between failures and g be the cost for generating a checkpoint. A result in the literature shows that when M is much larger than g and totally reliable checkpoints are to be placed uniformly every w units of time, a first order approximation to the optimum reliable checkpoint interval is $w = \sqrt{2gM}$ [76]. This result is extended to the case of unreliable checkpoints with $q < 1$, where it is shown that $w = \sqrt{2qgM/(2-q)}$ is a first order approximation to the optimum faulty checkpoint interval.

Chapter 2

Assigning Tasks for Efficiency in Hadoop

Everyone has his own specific vocation or mission in life; everyone must carry out a concrete assignment that demands fulfillment.

– Viktor E. Frankl

2.1 Introduction

2.1.1 The Hadoop Parallel Computing Platform

Apache Hadoop [10] is a free Java implementation of Google's MapReduce [22, 23, 54] in the open source software community. After a few years' development, it has become the most popular cloud computing platform for many large data processing applications.

Researchers in academia have adapted Hadoop to several different architectures. For example, Ranger et al. [67] evaluate MapReduce in multi-core and multi-processor systems, Kruijf et al. [20] implement MapReduce on the Cell B.E. processor architecture, and He et al. [45] propose a MapReduce framework on graphics processors. Many related applications using Hadoop have also been developed to solve various practical problems.

Database vendors from industry have developed many ways of combining Hadoop parallel processing engine with the traditional database management systems, providing connectors to allow external Hadoop programs to retrieve data from databases and to s-tore Hadoop output in databases [63, 66]. Aster Data provides MapReduce extensions to SQL called SQL-MapReduce for writing MapReduce programs within the database [31]. There are several parallel execution systems built on top of Hadoop [3, 57, 64, 74]. Usually queries are compiled to a sequence of MapReduce jobs and these jobs are then executed on a Hadoop cluster. Hybrid systems have been built that either give Hadoop access to traditional DBMS [72], or allow Hadoop to coordinate parallel queries over many traditional DBMS instances [43], or provide Hadoop interface inside database resident JVM and rely on database parallel query engine to execute the actual computation [73].

2.1.2 The MapReduce Framework

A Hadoop system runs on top of a distributed file system, called the Hadoop Distributed File System. HDFS usually runs on networked commodity PCs, where data are replicated and locally stored on hard disks of each machine. To store and process huge volumes of data sets, HDFS typically uses a block size of 64MB.

In the MapReduce framework, a job splits the input data into independent blocks, which are processed by the *map* tasks in parallel. Each map task processes a single block[1] consisting of some number of records. Each record in turn consists of a key/value pair. A map task applies the user defined map function to each input key/value pair and pro-

[1] Strictly speaking, a map task in Hadoop sometimes processes data that comes from two successive file blocks. This occurs because file blocks do not respect logical record boundaries, so the last logical record processed by a map task might lie partly in the current data block and partly in the succeeding block, requiring the map task to access the succeeding block in order to fetch the tail end of its last logical record.

duces intermediate key/value pairs. The framework then sorts the intermediate data, and forwards them to the *reduce* tasks via interconnected networks. After receiving all intermediate key/value pairs with the same key, a reduce task executes the user defined reduce function and produces the output data. Then the output data is written back to the HDFS.

In such a framework, there is a single server, called the *master*[2], that keeps track of all jobs in the whole distributed system. The master runs a special process, called the *jobtracker*, that is responsible for task assignment and scheduling for the whole system. For the rest of servers that are called the *slaves*, each of them runs a process called the *tasktracker*. The tasktracker schedules the several tasks assigned to the single server in a way similar to a normal operating system.

The map task assignment is a vital part that affects the completion time of the whole job. Each reduce task cannot begin until it receives the required intermediate data from all finished map tasks. For map/reduce tasks, the time to load data often dominates the actual computation time. A task assigned to a machine not possessing the needed input data must obtain it over the network from a remote machine. This can considerably increase the time to complete the task. Therefore, moving computation close to the data is a design goal in the MapReduce framework.

2.1.3 Related Work

Since Kuhn [51] presented the first method for the classic assignment problem in 1955, variations of the assignment problem have been under extensive study in many areas [13]. In the classic assignment problem, there are identical number of jobs and persons. An assignment is a one-to-one mapping from tasks to persons. Each job introduces a cost when it is assigned to a person. The objective is to find an optimal assignment that minimizes the total cost over all persons.

[2]There might be some shadow masters as backup in case when the master node fails.

In the area of parallel and distributed computing, when jobs are processed in parallel over several machines, one is interested in balancing load so as to minimize the job completion time. This problem is sometimes called the minimum makespan scheduling problem and in general is known to be $\mathcal{NP}$-hard [34]. Under the identical-machine model, there are some well-known approximation algorithms. For example, Graham [39] presented a $(2-1/n)$-approximation algorithm in 1966, where n is the total number of machines. Graham [40] presented another $4/3$-approximation algorithm in 1969. However, under the unrelated-machine model, this problem is known to be APX-hard, both in terms of its offline [55] and online [4, 6] approximability. Randomized algorithms are also powerful tools for load balancing over parallel machines. The power of two choices phenomenon [5, 59] is one of the most significant results in recent development. A simple randomized algorithm selects a random machine and assign the task to it. However, by selecting two random machines and assign the task to the less loaded one, it is shown that exponential improvements in makespan can be achieved. Readers are referred to [60] for a thorough review of such development.

The scheduling mechanisms and policies that assign tasks to servers within the MapReduce framework can have a profound effect on efficiency [8, 10]. An early version of Hadoop uses a simple heuristic algorithm that greedily exploits data locality. Zaharia, Konwinski and Joseph [77] present some heuristic refinements based on experimental results. Moseley et al. [61] formalize job scheduling in MapReduce as a generalization of the two-stage flexible flow shop problem and presented approximation algorithms for flow time minimization. Some results in this chapter are originally presented in [28, 29].

2.1.4 Our Contributions

In this chapter, we present an algorithmic study of the task assignment problems in the Hadoop/MapReduce framework. We present a mathematical model to evaluate the cost of task assignments in Section 2.2. Based on this model, we show that it is infeasible to find the optimal assignment unless $\mathcal{P} = \mathcal{NP}$. Theorem 2.1 shows that the task assignment problem in Hadoop remains hard even if all servers have equal capacity of 3 units, the cost function only has 2 values in its range, and each data block has at most 2 replicas.

In Section 2.4, we analyze a simple round robin algorithm for the uniform HTA problem. Theorem 2.11 reveals that the intuition is wrong that increasing the number of replicas always helps load balancing. Using round robin task assignment, adding more replicas into the system can sometimes result in worse maximum load. Theorems 2.12 and 2.13 show there is a multiplicative gap in maximum load between the optimal assignment and the assignment computed by round robin in the worst case.

In Section 2.5, we present Algorithm 2 for the general HTA problem. This algorithm employs maximum flow and increasing threshold techniques. Theorem 2.17 shows that the assignments computed by Algorithm 2 are optimal to within an additive constant that depends only on the number of servers and the remote cost function. Theorem 2.25 presents a matching lower bound to Theorem 2.17 on a restricted class of problem instances.

2.2 Problem Formalization

Definition 2.1 *A* Map-Reduce schema (MR-schema) *is a pair* (T, S)*, where* T *is a set of* tasks *and* S *is a set of* servers*. Let* $m = |T|$ *and* $n = |S|$*. A* task assignment *is a function* $A\colon T \to S$ *that assigns each task* t *to a server* $A(t)$*.*[3] *Let* $\mathcal{A} = \{T \to S\}$ *be the set of all possible task assignments. An* MR-system *is a triple* (T, S, w)*, where* (T, S) *is an MR-schema and* $w : T \times \mathcal{A} \to \mathbb{R}^+$ *is a* cost function*.*

Intuitively, $w(t, A)$ is the time to perform task t on server $A(t)$ in the context of the

[3] In an MR-schema, it is common that $|T| \geq |S|$. Therefore in this chapter, unlike the classic assignment problem where an assignment refers to a *one-to-one* mapping or a *permutation* [13, 51], we instead use the notion of *many-to-one* mapping.

complete assignment A. The motivation for this level of generality is that the time to execute a task t in Hadoop depends not only on the task and the server speed, but also on possible network congestion, which in turn is influenced by the other tasks running on the cluster.

Definition 2.2 *The* load *of server s under assignment A is defined as $L_s^A = \sum_{t:A(t)=s} w(t, A)$. The* maximum load *under assignment A is defined as $L^A = \max_s L_s^A$. The* total load *under assignment A is defined as $H^A = \sum_s L_s^A$.*

An MR-system models a cloud computer where all servers work in parallel. Tasks assigned to the same server are processed sequentially, whereas tasks assigned to different servers run in parallel. Thus, the total completion time of the cloud under task assignment A is given by the maximum load L^A.

Our notion of an MR-system is very general and admits arbitrary cost functions. To usefully model Hadoop as an MR-system, we need a realistic but simplified cost model.

In Hadoop, the cost of a map task depends frequently on the location of its data. If the data is on the server's local disk, then the cost (execution time) is considerably lower than if the data is located remotely and must be fetched across the network before being processed.

We make several simplifying assumptions. We assume that all tasks and all servers are identical, so that for any particular assignment of tasks to servers, all tasks whose data are locally available take the same amount of time w_{loc}, and all tasks whose data are remote take the same amount of time w_{rem}. However, we do not assume that w_{rem} is constant over all assignments. Rather, we let it grow with the total number of tasks whose data are remote. This reflects the increased data fetch time due to overall network congestion. Thus, $w_{\text{rem}}(r)$ is the cost of each remote task in every assignment with exactly r remote tasks. We assume that $w_{\text{rem}}(r) \geq w_{\text{loc}}$ for all r and that $w_{\text{rem}}(r)$ is (weakly) monotone increasing in r.

We formalize these concepts below. In each of the following, (T, S) is an MR-schema.

Definition 2.3 *A* data placement *is a relation $\rho \subseteq T \times S$ such that for every task $t \in T$, there exists at least one server $s \in S$ such that $\rho(t, s)$ holds.*

The placement relation describes where the input data blocks are placed. If $\rho(t, s)$ holds, then server s locally stores a replica of the data block that task t needs.

Definition 2.4 *We represent the placement relation ρ by an unweighted bipartite graph, called the* placement graph. *In the placement graph $G_\rho = ((T, S), E)$, T consists of m task nodes and S consists of n server nodes. There is an edge $(t, s) \in E$ iff $\rho(t, s)$ holds.*

Definition 2.5 *A* partial assignment *α is a partial function from T to S. We regard a partial assignment as a set of ordered pairs with pairwise distinct first elements, so for partial assignments β and α, $\beta \supseteq \alpha$ means β* extends *α. If $s \in S$, the* restriction of *α* to *s is the partial assignment $\alpha|_s = \alpha \cap (T \times \{s\})$. Thus, $\alpha|_s$ agrees with α for those tasks that α assigns to s, but all other tasks are unassigned in $\alpha|_s$.*

Definition 2.6 *Let ρ be a data placement and β be a partial assignment. A task $t \in T$ is* local in *β if $\beta(t)$ is defined and $\rho(t, \beta(t))$. A task $t \in T$ is* remote in *α if $\beta(t)$ is defined and $\neg\rho(t, \beta(t))$. Otherwise t is* unassigned in *β. Let ℓ^β, r^β and u^β be the number of local tasks, remote tasks, and unassigned tasks in β, respectively. For any $s \in S$, let ℓ_s^β be the number of local tasks assigned to s by β. Let $k^\beta = \max_{s \in S} \ell_s^\beta$.*

Definition 2.7 *Let ρ be a data placement, A be an assignment, $w_{\text{loc}} \in \mathbb{R}^+$, and $w_{\text{rem}} : \mathbb{N} \to \mathbb{R}^+$ such that $w_{\text{loc}} \leq w_{\text{rem}}(0) \leq w_{\text{rem}}(1) \leq w_{\text{rem}}(2) \ldots$. Let $w_{\text{rem}}^A = w_{\text{rem}}(r^A)$. The*

Hadoop cost function *with parameters ρ, w_{loc}, and $w_{\text{rem}}(\cdot)$ is the function w defined by*

$$w(t, A) = \begin{cases} w_{\text{loc}} & \textit{if } t \textit{ is local in } A, \\ w_{\text{rem}}^{A} & \textit{otherwise.} \end{cases}$$

We call ρ the placement *of w, and w_{loc} and $w_{\text{rem}}(\cdot)$ the* local *and* remote *costs of w, respectively.*

The definition of Hadoop cost function allows positive real costs. However, when considering algorithm complexity, one needs to deal with the issues of number representation and precision. We focus on rational costs in the following discussion. Since ρ is encoded by mn bits, w_{loc} is encoded by one rational number, and $w_{\text{rem}}(\cdot)$ is encoded by $m + 1$ rational numbers, the Hadoop cost function $w(\rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$ is encoded by mn bits plus $m + 2$ rational numbers.

Definition 2.8 *A* Hadoop MR-system (HMR-system) *is the MR-system (T, S, w), where w is the Hadoop cost function with parameters ρ, w_{loc}, and $w_{\text{rem}}(\cdot)$. A HMR-system is defined by $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.*

Problem 2.1 Hadoop Task Assignment Problem (HTA)

1. **Instance:** An HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.
2. **Objective:** Find an assignment A that minimizes L^A.

Sometimes the cost of running a task on a server only depends on the placement relation and its data locality, but not on the assignment of other tasks.

Definition 2.9 *A Hadoop cost function w is called* uniform *if $w_{\text{rem}}(r) = c$ for some constant c and all $r \in \mathbb{N}$. A* uniform HMR-system (UHMR-system) *is an HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$, where w is uniform.*

Problem 2.2 Uniform Hadoop Task Assignment Problem (UHTA)

1. **Instance:** A UHMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.
2. **Objective:** Find an assignment A that minimizes L^A.

The number of replicas of each data block may be bounded, often by a small number such as 2 or 3.

Definition 2.10 *Call a placement graph $G = ((T, S), E)$* j-replica-bounded *if the degree of t is at most j for all $t \in T$. A* j-replica-bounded-UHMR-system (j-UHMR-system) *is a UHMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$, where G_ρ is j-replica-bounded.*

Problem 2.3 j-Uniform Hadoop Task Assignment Problem (j-UHTA)

1. **Instance:** A j-UHMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.
2. **Objective:** Find an assignment A that minimizes L^A.

2.3 Hardness of Task Assignment

In this section, we analyze the hardness of the various HTA optimization problems by showing the corresponding decision problems to be $\mathcal{NP}$-complete.

2.3.1 Task Assignment Decision Problems

Definition 2.11 *Given a server* capacity k*, a task assignment A is* k-feasible *if $L^A \leq k$. An HMR-system is* k-admissible *if there exists a k-feasible task assignment.*

The decision problem corresponding to a class of HMR-systems and capacity k asks whether a given HMR-system in the class is k-admissible. Thus, the k-HTA problem asks about arbitrary HMR-systems, the k-UHTA problem asks about arbitrary UHMR-systems, and the k-j-UHTA problem (which we write (j,k)-UHTA) asks about arbitrary j-UHMR-systems.

2.3.2 $\mathcal{NP}$-completeness of (2,3)-UHTA

The (2,3)-UHTA problem is a very restricted subclass of the general k-admissibility problem for HMR-systems. In this section, we restrict even further by taking $w_{\text{loc}} = 1$ and $w_{\text{rem}} = 3$. This problem represents a simple scenario where the cost function assumes only the two possible values 1 and 3, each data block has at most 2 replicas, and each server has capacity 3. Despite its obvious simplicity, we show that (2,3)-UHTA is $\mathcal{NP}$-complete. It follows that all of the less restrictive decision problems are also $\mathcal{NP}$-complete, and the corresponding optimization problems do not have feasible solutions unless $\mathcal{P} = \mathcal{NP}$.

Theorem 2.1 $(2,3)$-UHTA *with costs* $w_{\text{loc}} = 1$ *and* $w_{\text{rem}} = 3$ *is* $\mathcal{NP}$*-complete.*

The proof method is to construct a polynomial-time reduction from 3SAT to (2,3)-UHTA. Let $\mathcal{G}$ be the set of all 2-replica-bounded placement graphs. Given $G_\rho \in \mathcal{G}$, we define the HMR-system $\mathcal{M}_G = (T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$, where $w_{\text{loc}} = 1$ and $w_{\text{rem}}(r) = 3$ for all r. We say that G is 3-*admissible* if $\mathcal{M}_G$ is 3-admissible. We construct a polynomial-time computable mapping $f : \text{3CNF} \to \mathcal{G}$, and show that a 3CNF formula ϕ is satisfiable iff $f(\phi)$ is 3-admissible. We shorten "3-admissible" to "admissible" in the following discussion.

We first describe the construction of f. Let $\phi = C_1 \wedge C_2 \cdots \wedge C_\alpha$ be a 3CNF formula, where each $C_u = (l_{u1} \vee l_{u2} \vee l_{u3})$ is a clause and each l_{uv} is a literal. Let $x_1, \cdots, x_\beta$ be the variables that appear in ϕ. Therefore, ϕ contains exactly 3α instances of literals, each of which is either x_i or $\neg x_i$, where $i \in [1, \beta]$.[4] Let ω be the maximum number of occurrences

[4]The notation [a,b] in our discussion represents the set of integers $\{a, a+1, \cdots, b-1, b\}$.

of any literal in ϕ. Table 2.1 summarizes the parameters of ϕ.

Table 2.1: Parameters of the 3CNF ϕ

clauses (C_u)	α	variables (v_i)	β
literals (l_{uv})	3α	max-occur of any literal	ω

For example, in $\phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_4 \vee x_5) \wedge (\neg x_1 \vee x_4 \vee \neg x_6)$, we have $\alpha = 3$, $\beta = 6$, and $\omega = 2$ since x_1 occurs twice.

Given ϕ, we construct the corresponding placement graph G which comprises several disjoint copies of the three types of gadget described below, connected together with additional edges.

The first type of gadget is called a *clause gadget*. Each clause gadget u contains a *clause server* C_u, three *literal tasks* l_{u1}, l_{u2}, l_{u3} and an *auxiliary task* a_u. There is an edge between each of these tasks and the clause server. Since ϕ contains α clauses, G contains α clause gadgets. Thus, G contains α clause servers, 3α literal tasks and α auxiliary tasks. Figure 2.1 describes the structure of the u-th clause gadget. We use circles and boxes to represent tasks and servers, respectively.

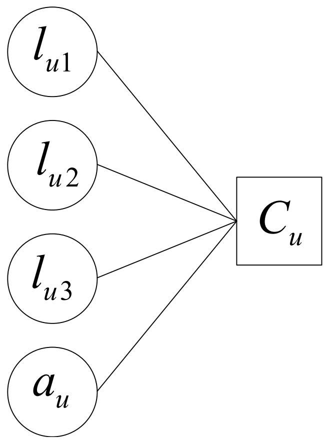

Figure 2.1: The structure of the u-th clause gadget.

The second type of gadget is called a *variable gadget*. Each variable gadget contains 2ω *ring servers* placed around a circle. Let $R_j^{(i)}$ denote the server at position $j \in [1, 2\omega]$

in ring i. Define the set $\mathcal{T}_i$ to be the servers in odd-numbered positions. Similarly, define the set $\mathcal{F}_i$ to be the servers in even-numbered positions. Between each pair of ring servers $R_j^{(i)}$ and $R_{j+1}^{(i)}$, we place a *ring task* $r_j^{(i)}$ connected to its two neighboring servers. To complete the circle, $r_{2\omega}^{(i)}$ is connected to $R_{2\omega}^{(i)}$ and $R_1^{(i)}$. There are also ω *variable tasks* $v_j^{(i)} : j \in [1, \omega]$ in ring i, but they do not connect to any ring server. Since ϕ contains β variables, G contains β variable gadgets. Thus, G contains $2\beta\omega$ ring servers, $2\beta\omega$ ring tasks and $\beta\omega$ variable tasks. Figure 2.2 describes the structure of the i-th variable gadget.

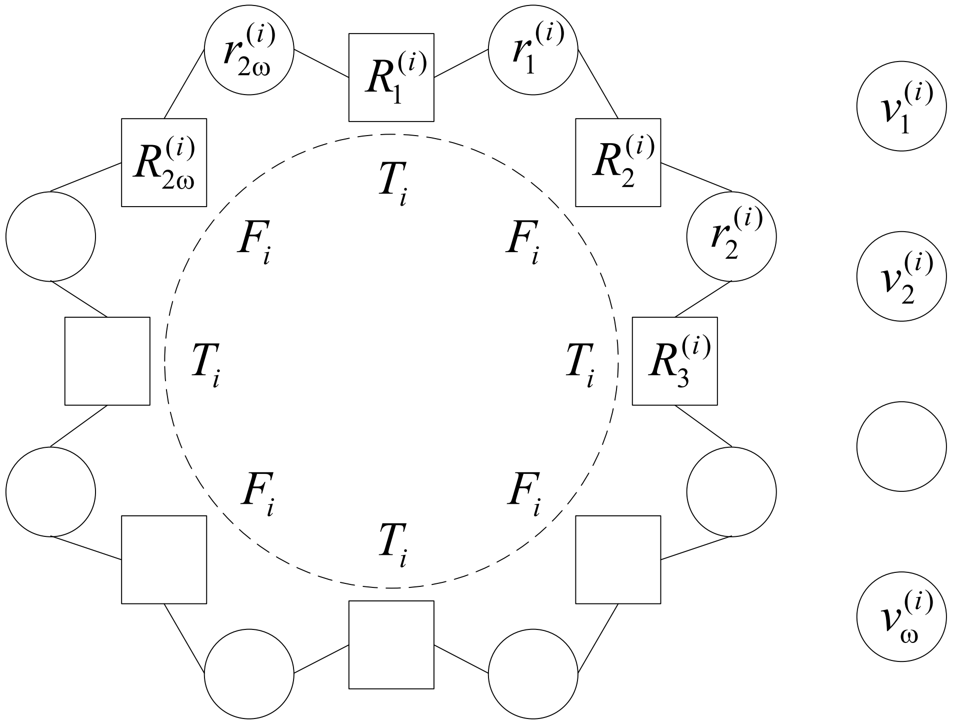

Figure 2.2: The structure of the i-th variable gadget.

The third type of gadget is called a *sink gadget*. The sink gadget contains a *sink server* P and three *sink tasks* p_1, p_2, p_3. Each sink task is connected to the sink server. G only contains one sink gadget. Figure 2.3 describes the structure of the sink gadget.

There are also some inter-gadget edges in G. We connect each variable task $v_j^{(i)}$ to the sink server P. We also connect each literal task l_{uv} to a unique ring server $R_j^{(i)}$. To be more precise, if literal l_{uv} is the j-th occurrence of x_i in ϕ, connect the literal task l_{uv} to

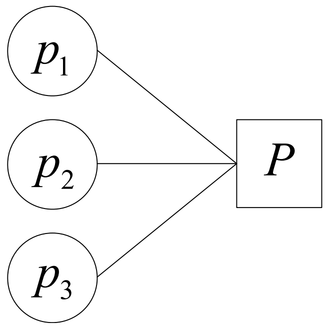

Figure 2.3: The structure of the sink gadget.

ring server $R^{(i)}_{2j-1} \in \mathcal{T}_i$; if literal l_{uv} is the j-th occurrence of $\neg x_i$ in ϕ, connect the literal task l_{uv} to ring server $R^{(i)}_{2j} \in \mathcal{F}_i$. These inter-gadget edges complete the graph G. Table 2.2 summarizes the parameters of G.

Table 2.2: Parameters of the HMR-graph G

clause server C_u	α	literal task l_{uv}	3α
auxiliary task a_u	α	ring server $R^{(i)}_j$	$2\beta\omega$
ring task $r^{(i)}_j$	$2\beta\omega$	variable task $v^{(i)}_j$	$\beta\omega$
sink server P	1	sink task p_j	3

Lemma 2.2 *For any $\phi \in$ 3CNF, the graph $f(\phi)$ is 2-replica-bounded.*

Proof. We count the number of edges from each task node in $f(\phi)$. Each clause task has 2 edges, each auxiliary task has 1 edge, each ring task has 2 edges, each variable task has 1 edge, and each sink task has 1 edge. Therefore, $f(\phi)$ is 2-replica-bounded. $\square$

The following lemma is immediate.

Lemma 2.3 *The mapping f : 3CNF $\rightarrow \mathcal{G}$ is polynomial-time computable.*

Lemma 2.4 *If ϕ is satisfiable, then $G = f(\phi)$ is admissible.*

Proof. Let σ be a satisfying truth assignment for ϕ, and we construct a feasible assignment A in $G = f(\phi)$. First of all, assign each sink task to the sink server, i.e., let $A(p_i) = P$

for all $i \in [1,3]$. Then assign each auxiliary task a_u to the clause server C_u, i.e., let $A(a_u) = C_u$ for all $u \in [1,\alpha]$. If $\sigma(x_i) = \mathit{true}$, then assign ring tasks $r_j^{(i)} : j \in [1, 2\omega]$ to ring servers in $\mathcal{T}_i$, variable tasks $v_j^{(i)} : j \in [1,\omega]$ to ring servers in $\mathcal{F}_i$. If $\sigma(x_i) = \mathit{false}$, then assign ring tasks $r_j^{(i)} : j \in [1, 2\omega]$ to ring servers in $\mathcal{F}_i$, variable tasks $v_j^{(i)} : j \in [1,\omega]$ to ring servers in $\mathcal{T}_i$. If literal $l_{uv} = x_i$ and $\sigma(x_i) = \mathit{true}$, then assign task l_{uv} to its local ring server in $\mathcal{T}_i$. If literal $l_{uv} = \neg x_i$ and $\sigma(x_i) = \mathit{false}$, then assign task l_{uv} to its local ring server in $\mathcal{F}_i$. Otherwise, assign task l_{uv} to its local clause server C_u.

We then check this task assignment is feasible. Each ring server is assigned either at most three local tasks (two ring tasks and one literal task), or one remote variable task. In either case, the load does not exceed the capacity 3. The number of tasks assigned to each clause server C_u is exactly the number of false literals in C_u under σ plus one (the auxiliary task), and each task is local to C_u. Thus, the load is at most 3. The sink server is assigned three local sink tasks and the load is exactly 3. Therefore, all constraints are satisfied and A is feasible. This completes the proof of Lemma 2.4. $\square$

The proof of the converse of Lemma 2.4 is more involved. The method is given a feasible assignment A in $G = f(\phi)$, we first construct a feasible assignment B in G such that $B(t) \neq P$ for all $t \in T - \{p_1, p_2, p_3\}$. Then we remove the sink tasks and the sink server from further consideration and consider the resulting graph G'. After that, we partition G' into two subgraphs, and construct a feasible assignment B' such that no tasks from one partition are remotely assigned to servers in the other partition. This step involves a case analysis. Finally, a natural way of constructing the satisfying truth assignment for ϕ follows.

Lemma 2.5 *Let A be a feasible task assignment. Then there exists a feasible task assignment B such that $B(t) \neq P$ for all $t \in T - \{p_1, p_2, p_3\}$.*

Proof. When A satisfies that $A(t) \neq P$ for all $t \in T - \{p_1, p_2, p_3\}$, let $B = A$. Otherwise, assume there exists a task t' such that $A(t') = P$ and $t' \in T - \{p_1, p_2, p_3\}$. Since the capacity of P is 3, there is at least one sink task, say p_1, is not assigned to P. Let $A(p_1) = Q$. Since $\rho(p_1, Q)$ does not hold, Q has only been assigned p_1 and $L_Q^A = 3$. Let $B(p_1) = P$ and $B(t') = Q$. Repeat the same process for all tasks other than p_1, p_2, p_3 that are assigned to P in A. Then let $B(t) = A(t)$ for the remaining tasks $t \in T$. To see B is feasible, note that $L_s^B \leq L_s^A \leq 3$ for all servers $s \in S$. □

Let G' be the subgraph induced by $(T - \{p_1, p_2, p_3\}, S - \{P\}) = (T', S')$. We have the following lemma.

Lemma 2.6 *Let A be a feasible task assignment in G. Then there exists a feasible task assignment A' in G'.*

Proof. Given A, Lemma 2.5 tells us that there exists another feasible assignment B in G such that $B(t) \neq P$ for all $t \in T'$. Let $A'(t) = B(t)$ for all $t \in T'$. Then A' is an assignment in G' since $A'(t) \in S - \{P\}$ for all $t \in T'$. To see A' is feasible, note that $L_s^{A'} \leq L_s^B \leq 3$ for all servers $s \in S'$. □

We further partition G' into two subgraphs G_C and G_R. G_C is induced by nodes $\{C_u : u \in [1, \alpha]\} \cup \{a_u : u \in [1, \alpha]\} \cup \{l_{uv} : u \in [1, \alpha], v \in [1, 3]\}$ and G_R is induced by nodes $\{R_j^{(i)} : i \in [1, \beta], j \in [1, 2\omega]\} \cup \{r_j^{(i)} : i \in [1, \beta], j \in [1, 2\omega]\} \cup \{v_j^{(i)} : i \in [1, \beta], j \in [1, \omega]\}$. In other words, G_C consists of all clause gadgets while G_R consists of all variable gadgets.

If a task in one partition is remotely assigned to a server in the other partition, we call this task a *cross-boundary* task. Let n_c^A be the number of cross-boundary tasks that are in G_C and assigned to servers in G_R by A, n_r^A be the number of cross-boundary tasks that are in G_R and assigned to servers in G_C by A. We have the following lemmas.

Lemma 2.7 *Let A be a feasible assignment in G' such that $n_c^A > 0$ and $n_r^A > 0$. Then there exist a feasible assignment B in G' such that one of n_c^B and n_r^B equals $|n_c^A - n_r^A|$ and the other one equals 0.*

Proof. Assume $t_i \in G_C$, $s_i \in G_R$ and $A(t_i) = s_i$; $t_i' \in G_R$, $s_i' \in G_C$ and $A(t_i') = s_i'$. Then each of s_i and s_i' is assigned one remote task. Let $B(t_i) = s_i'$ and $B(t_i') = s_i$, and then $L_{s_i}^B \leq L_{s_i}^A = 3$ and $L_{s_i'}^B \leq L_{s_i'}^A = 3$. This process decreases n_c and n_r each by one, and the resulting assignment is also feasible. Repeat the same process until the smaller one of n_c and n_r becomes 0. Then let $B(t) = A(t)$ for all the remaining tasks $t \in T'$. It is obvious that B is feasible, and one of n_c^B and n_r^B equals $|n_c^A - n_r^A|$ and the other one equals 0. $\square$

Lemma 2.8 *Let A be a feasible assignment in G' such that $n_c^A = 0$. Then $n_r^A = 0$.*

Proof. For the sake of contradiction, assume $t_i \in G_R$, $s_i \in G_C$ and $A(t_i) = s_i$. For each server $s_j \in G_C$, there is one auxiliary task $a_u : u \in [1, \alpha]$ such that $\rho(a_u, s_j)$ holds. Since $w_{\text{loc}} = 1$ and $w_{\text{rem}} = 3$, if A is feasible then $A(a_u) \neq A(a_v)$ for $u \neq v$. Since there are α auxiliary tasks and α servers in G_C, one server is assigned exactly one auxiliary task. Since $A(t_i) = s_i$, $L_{s_i}^A \geq 1 + 3 > 3$, contradicting the fact that A is feasible. Therefore, there is no $t_i \in G_R$ and $s_i \in G_C$ such that $A(t_i) = s_i$. Thus, $n_r^A = 0$. $\square$

Lemma 2.9 *Let A be a feasible assignment in G' such that $n_r^A = 0$. Then $n_c^A = 0$.*

Proof. For the sake of contradiction, assume $t_i \in G_C$, $s_i \in G_R$ and $A(t_i) = s_i$. Let k_0, k_1, k_2, k_3 denote the number of ring servers filled to load $0, 1, 2, 3$, respectively. From the total number of servers in G_R, we have

$$k_0 + k_1 + k_2 + k_3 = 2\beta\omega \tag{2.1}$$

Similarly, from the total number of tasks in G_R, we have

$$0 \cdot k_0 + 1 \cdot k_1 + 2 \cdot k_2 + 1 \cdot k_3 = 3\beta\omega \tag{2.2}$$

Subtracting (2.1) from (2.2) gives $k_2 = \beta\omega + k_0$. Assigning both neighboring ring tasks to the same ring server fills it to load 2. Since there are only $2\beta\omega$ ring servers, we have $k_2 \leq \beta\omega$. Hence, $k_0 = 0$ and $k_2 = \beta\omega$. This implies that all ring tasks are assigned to ring servers in alternating positions in each ring.

There are $\beta\omega$ remaining ring servers and $\beta\omega$ variable tasks. Therefore, a variable task is remotely assigned to one of the remaining ring servers by A.

Now consider the server s_i that has been remotely assigned $t_i \in G_C$. If it is assigned two ring tasks, its load is $L_{s_i}^A = 2 + 3 > 3$. If it is assigned one variable task, its load is $L_{s_i}^A = 3 + 3 > 3$. A is not feasible in either case. Therefore, there is no $t_i \in G_C$ and $s_i \in G_R$ such that $A(t_i) = s_i$. Thus, $n_c^A = 0$. □

Now we prove the following Lemma.

Lemma 2.10 *If $G = f(\phi)$ is admissible, then ϕ is satisfiable.*

Proof. Given feasible task assignment A in $G = f(\phi)$, we construct the satisfying truth assignment σ for ϕ. From Lemmas 2.6, 2.7, 2.8 and 2.9, we construct a feasible assignment B in G', such that $n_c^B = n_r^B = 0$, and in each variable gadget i, either servers in $\mathcal{T}_i$ or servers in $\mathcal{F}_i$ are saturated by variable tasks. If ring servers in $\mathcal{F}_i$ are saturated by variable tasks, let $\sigma(x_i) = \mathit{true}$. If ring servers in $\mathcal{T}_i$ are saturated by variable tasks, let $\sigma(x_i) = \mathit{false}$.

To check that this truth assignment is a satisfying assignment, note that for the three literal tasks l_{u1}, l_{u2}, l_{u3}, at most two of them are assigned to the clause server C_u. There must be one literal task, say l_{uv}, that is locally assigned to a ring server. In this case,

$\sigma(l_{uv}) = true$ and thus the clause $\sigma(C_u) = true$. This fact holds for all clauses and thus indicates that $\sigma(\phi) = \sigma(\bigwedge C_u) = true$. This completes the proof of Lemma 2.10. □

Finally we prove the main theorem.

Proof of Theorem 2.1: Lemmas 2.3, 2.4 and 2.10 establish that 3SAT $\leq_p$ (2,3)-UHTA via f. Therefore, (2,3)-UHTA is $\mathcal{NP}$-hard. It is easy to see that (2,3)-UHTA $\in \mathcal{NP}$ because in time $O(mn)$ a nondeterministic Turing machine can guess the assignment and accept iff the maximum load under the assignment does not exceed 3. Therefore, $(2,3)$-UHTA is $\mathcal{NP}$-complete. □

2.4 A Round Robin Algorithm

In this section, we analyze a simple round robin algorithm for the UHTA problem. Algorithm 1 is inspired by the Hadoop scheduler algorithm. It scans over each server in a round robin fashion. When assigning a new task to a server, Algorithm 1 tries heuristically to exploit data locality. Since we have not specified the order of assigned tasks, Algorithm 1 may produce many possible outputs (assignments).

Algorithm 1: The round robin algorithm exploring locality.

```
Input: a set of unassigned tasks $T$, a list of servers $\{s_1, s_2, \cdots, s_n\}$, a placement
       relation $\rho$
1 define $i \leftarrow 1$ as an index variable
2 define $A$ as an assignment
3 $A(t) \leftarrow \perp$ (task $t$ is unassigned) for all $t$
4 while exists unassigned task do
5     if exists unassigned task $t$ such that $\rho(t, s_i)$ holds then
6         update $A$ by assigning $A(t) \leftarrow s_i$
7     else
8         pick any unassigned task $t'$, update $A$ by assigning $A(t') \leftarrow s_i$
9     $i \leftarrow (i \bmod n) + 1$
Output: assignment $A$
```

Algorithm 1 is analogous to the Hadoop scheduler algorithm up to core version 0.19. There are three differences, though. First, the Hadoop algorithm assumes three kinds of placement: data-local, rack-local and rack-remote, whereas Algorithm 1 assumes only t-wo: local and remote. Second, the Hadoop scheduler works incrementally rather than assigning all tasks initially. Last, the Hadoop algorithm is deterministic, whereas Algorithm 1 is nondeterministic.

Theorem 2.11 *If $w_{\text{rem}} > w_{\text{loc}}$, increasing the number of data block replicas may increase the maximum load of the assignment computed by Algorithm 1.*

Proof. The number of edges in the placement graph is equal to the number of data block replicas, and thus adding a new edge in the placement graph is equivalent to adding a new replica in the system. Consider the simple placement graph G where $m = n$, and there is an edge between task t_i and s_i for all $1 \leq i \leq n$. Running Algorithm 1 gives an assignment A in which task t_i is assigned to s_i for all $1 \leq i \leq n$, and thus $L^A = w_{\text{loc}}$. Now we add one edge between task t_n and server s_1. We run Algorithm 1 on this new placement graph G' to get assignment A'. It might assign task t_n to server s_1 in the first step. Following that, it assigns t_i to s_i for $2 \leq i \leq n-1$, and it finally assigns t_1 to s_n. Since t_1 is remote to s_n, this gives $L^{A'} = w_{\text{rem}}$. Therefore $L^{A'} > L^A$. $\square$

Theorem 2.11 indicates that increasing the number of data block replicas is not always beneficial for Algorithm 1. In the remaining part of this section, we show that the assignments computed by Algorithm 1 might deviate from the optimum by a multiplicative factor. In the following, let O be an assignment that minimizes L^O.

Theorem 2.12 *Let A be an assignment computed by Algorithm 1. Then $L^A \leq (w_{\text{rem}}/w_{\text{loc}}) \cdot L^O$.*

Proof. On the one hand, pigeonhole principle says there is a server assigned at least $\lceil m/n \rceil$ tasks. Since the cost of each task is at least w_{loc}, the load of this server is at least

$\lceil m/n \rceil \cdot w_{\text{loc}}$. Thus, $L^O \geq \lceil m/n \rceil \cdot w_{\text{loc}}$. On the other hand, Algorithm 1 runs in a round robin fashion where one task is assigned at a time. Therefore, the number of tasks assigned to each server is at most $\lceil m/n \rceil$. Since the cost of each task is at most w_{rem}, the load of a server is at most $\lceil m/n \rceil \cdot w_{\text{rem}}$. Thus, $L^A \leq \lceil m/n \rceil \cdot w_{\text{rem}}$. Combining the two, we have $L^A \leq (w_{\text{rem}}/w_{\text{loc}}) \cdot L^O$. □

Theorem 2.13 *Let T and S be such that $m \leq n(n-2)$. There exist a placement ρ and an assignment A such that A is a possible output of Algorithm 1, $L^A \geq \lfloor m/n \rfloor \cdot w_{\text{rem}}$, and $L^O = \lceil m/n \rceil \cdot w_{\text{loc}}$.*

Proof. We prove the theorem by constructing a placement graph G_ρ. Partition the set T of tasks into n disjoint subsets $T_i : 1 \leq i \leq n$, such that $\lceil m/n \rceil \geq |T_i| \geq |T_j| \geq \lfloor m/n \rfloor$ for all $1 \leq i \leq j \leq n$. Now in the placement graph G_ρ, connect tasks in T_i to server s_i, for all $1 \leq i \leq n$. These set of edges guarantee that $L^O = \lceil m/n \rceil \cdot w_{\text{loc}}$. We then connect each task in T_n to a different server in the subset $S' = \{s_1, s_2, \cdots, s_{n-1}\}$. Since $m \leq n(n-2)$, we have $\lceil m/n \rceil \leq m/n + 1 \leq n - 1$, which guarantees $|S'| \geq |T_n|$. This completes the placement graph G_ρ. Now run Algorithm 1 on G_ρ. There is a possible output A where tasks in T_n are assigned to servers in S'. In that case, all tasks that are local to server s_n are assigned elsewhere, and thus s_n is assigned remote tasks. Since s_n is assigned at least $\lfloor m/n \rfloor$ tasks, this gives $L^A \geq \lfloor m/n \rfloor \cdot w_{\text{rem}}$. □

When $n \mid m$, the lower bound in Theorem 2.13 matches the upper bound in Theorem 2.12.

2.5 FLOWT: A Flow-based Algorithm

Theorem 2.1 shows that the problem of computing an optimal task assignment for the HTA problem is $\mathcal{NP}$-hard. Nevertheless, it is feasible to find task assignments whose load is at most an additive constant greater than the optimal load. We present such an algorithm in this section.

We first extend the definition of Hadoop cost function under complete assignments to partial assignments.

Definition 2.12 *Given a HMR-system* $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$*, let* α *be a partial assignment. Let* $w_{\text{rem}}^{\alpha} = w_{\text{rem}}(r^{v} + u^{\alpha})$*. Define the cost of task* t *under* α *to be*

$$w(t, \alpha) = \begin{cases} w_{\text{loc}} & \textit{if } t \textit{ is local in } \alpha, \\ w_{\text{rem}}^{\alpha} & \textit{otherwise.} \end{cases}$$

Let $K^{\alpha} = k^{\alpha} \cdot w_{\text{loc}}$.

The definition of remote cost under a partial assignment α is pessimistic. It assumes that tasks not assigned by α will eventually become remote, and each remote task will eventually have cost $w_{\text{rem}}(r^{\alpha} + u^{\alpha})$. This definition agrees with the definition of remote cost under a complete assignment A, because $u^{A} = 0$ and thus $w_{\text{rem}}^{A} = w_{\text{rem}}(r^{A} + u^{A}) = w_{\text{rem}}(r^{A})$.

For two partial assignments α and β such that $\beta \supseteq \alpha$, we define a new notation called *virtual load from* α below.

Definition 2.13 *For any task* t *and partial assignment* β *that extends* α*, let*

$$v^{\alpha}(t, \beta) = \begin{cases} w_{\text{loc}} & \textit{if } t \textit{ is local in } \beta, \\ w_{\text{rem}}^{\alpha} & \textit{otherwise.} \end{cases}$$

The virtual load of server s under β from α *is* $V_s^{\beta,\alpha} = \sum_{t:\beta(t)=s} v^{\alpha}(t, \beta)$*. The* maximum virtual load under β from α *is* $V^{\beta,\alpha} = \max_{s \in S} V_s^{\beta,\alpha}$.

Thus, v assumes pessimistically that tasks not assigned by β will eventually become remote, and each remote task will eventually have cost w_{rem}^{α}. When α is clear from context, we omit α and write $v(t, \beta)$, V_s^β and V^β, respectively. Note that $v^\alpha(t, \alpha) = w(t, \alpha)$ as in Definition 2.12.

Algorithm 2 works iteratively to produce a sequence of assignments and then outputs the best one, i.e., the one of least maximum server load. The iteration is controlled by an integer variable τ which is initialized to 1 and incremented on each iteration. Each iteration consists of two phases, *max-cover* and *bal-assign*:

- *Max-cover*: Given as input a placement graph G_ρ, an integer value τ, and a partial assignment α, max-cover returns a partial assignment α' of a subset T' of tasks, such that α' assigns no server more than τ tasks, every task in T' is local in α', and $|T'|$ is maximized over all such assignments. Thus, α' makes as many tasks local as is possible without assigning more than τ tasks to any one server. The name "max-cover" follows the intuition that we are actually trying to "cover" as many tasks as possible by their local servers, subject to the constraint that no server is assigned more than τ tasks.

- *Bal-assign*: Given as input a set of tasks T, a set of servers S, a partial assignment α computed by max-cover, and a cost function w, bal-assign uses a simple greedy algorithm to extend α to a complete assignment B by repeatedly choosing a server with minimal virtual load and assigning some unassigned task to it. This continues until all tasks are assigned. It thus generates a sequence of partial assignments $\alpha = \alpha_0 \subseteq \alpha_1 \subseteq \cdots \subseteq \alpha_u = B$, where $u = u^\alpha$. Every task t assigned in bal-assign contributes $v^\alpha(t, B) \le w_{\text{rem}}^\alpha$ to the virtual load of the server that it is assigned to. At the end, $w_{\text{rem}}^B \le w_{\text{rem}}^\alpha$, and equality holds only when $r^B = r^\alpha + u^\alpha$.

The astute reader might feel that it is intellectually attractive to use real server load as the criterion to choose servers in bal-assign because it embeds more accurate information. We do not know if this change ever results in a better assignment. We do know that it may require more computation. Whenever a local task is assigned, $r + u$ decreases by 1, so the remote cost $w_{\text{rem}}(r + u)$ may also decrease. If it does, the loads of all servers that have been assigned remote tasks must be recomputed. In the current version of the algorithm, we do not need to update virtual load when a local task is assigned because the virtual cost of remote tasks never changes in the course of bal-assign.

Algorithm 2: FLOWT: A flow-based algorithm for HTA.

```
  Input: an HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$
1 define $A, B$ as assignments
2 define $\alpha$ as a partial assignment
3 $\alpha(t) \leftarrow \perp$ (task $t$ is unassigned) for all $t$
4 for $\tau \leftarrow 1$ to $m$ do
5     $\alpha \leftarrow$ max-cover$(G_\rho, \tau, \alpha)$
6     $B \leftarrow$ bal-assign$(T, S, \alpha, w_{\text{loc}}, w_{\text{rem}}(\cdot))$
7 set $A$ equal to a $B$ with least maximum load
  Output: assignment $A$
```

2.5.1 Algorithm Description

We describe Algorithm 2 in greater detail here.

Max-cover

Max-cover (line 5 of Algorithm 2) augments the partial assignment $\alpha^{\tau-1}$ computed by the previous iteration to produce α^τ. (We define α^0 to be the empty partial assignment.) Thus, $\alpha^\tau \supseteq \alpha^{\tau-1}$, and α^τ maximizes the total number of local tasks assigned subject to the constraint that no server is assigned more than τ tasks in all.

The core of the max-cover phase is an augmenting path algorithm by Ford and Fulkerson [30]. The Ford-Fulkerson algorithm takes as input a network with edge capacities and an existing network flow, and outputs a maximum flow that respects the capacity constraints. A fact about this algorithm is well-known [18, 30].

Fact 2.14 *Given a flow network with integral capacities and an initial integral s-t flow f, the Ford-Fulkerson algorithm computes an integral maximum s-t flow f' in time $O(|E| \cdot (|f'| - |f|))$, where $|E|$ is the number of edges in the network and $|f|$ is the value of the flow f, i.e., the amount of flow passing from the source to the sink.*

During the max-cover phase at iteration τ, the input placement graph G_ρ is first converted to a corresponding flow network G'_ρ. G'_ρ includes all nodes in G_ρ and an extra source u and an extra sink v. In G'_ρ, there is an edge (u, t) for all $t \in T$ and an edge (s, v) for all $s \in S$. All of the original edges (t, s) in G_ρ remain in G'_ρ. The edge capacity is defined as follows: edge (s, v) has capacity τ for all $s \in S$, while all the other edges have capacity 1. Therefore, for any pair of (t, s), if there is a flow through the path $u \to t \to s \to v$, the value of this flow is no greater than 1. Then the input partial assignment α is converted into a network flow f_α as follows: if task t is assigned to server s in the partial assignment α, assign one unit of flow through the path $u \to t \to s \to v$.

The Ford-Fulkerson algorithm is then run on graph G'_ρ with flow f_α to find a maximum flow f'_α. From Fact 2.14, we know that the Ford-Fulkerson algorithm takes time $O(|E| \cdot (|f'_\alpha| - |f_\alpha|))$ in this iteration. This output flow f'_α at iteration τ will act as the input flow to the Ford-Fulkerson algorithm at iteration $\tau + 1$. The flow network at iteration $\tau + 1$ is the same as the one at iteration τ except that each edge (s, v) has capacity $\tau + 1$ for all $s \in S$. This incremental use of Ford-Fulkerson algorithm in successive iterations helps reduce the time complexity of the whole algorithm.

At the end of the max-cover phase, the augmented flow f'_α is converted back into a partial assignment α'. If there is one unit of flow through the path $u \to t \to s \to v$ in f'_α, we assign task t is to server s in α'. This conversion from a network flow to a partial assignment can always be done, because the flow is integral and all edges between tasks and servers have capacity 1. Therefore, there is a one-to-one correspondence between a unit flow through the path $u \to t \to s \to v$ and the assignment of task t to its local server s. It follows that $|f'_\alpha| = \ell^{\alpha'}$. By Fact 2.14, the Ford-Fulkerson algorithm computes a maximum flow that respects the capacity constraint τ. Thus, the following lemma is immediate.

Lemma 2.15 *Let α^τ be the partial assignment computed by max-cover at iteration τ, and β be any partial assignment such that $k^\beta \leq \tau$. Then $\ell^{\alpha^\tau} \geq \ell^\beta$.*

Bal-assign

Definition 2.14 *Let β and β' be partial assignments, t a task and s a server. We say that $\beta \xrightarrow{t:s} \beta'$* is a step that assigns t to s *if t is unassigned in β and $\beta' = \beta \cup \{(t, s)\}$. We say $\beta \to \beta'$* is a step, *if $\beta \xrightarrow{t:s} \beta'$ for some t and s.*

A sequence of steps $\alpha = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_u$ is a trace *if for each $i \in [1, u]$, if $\alpha_{i-1} \xrightarrow{t:s} \alpha_i$ is a step, then $V_s^{\alpha_{i-1},\alpha} \leq V_{s'}^{\alpha_{i-1},\alpha}$ for all $s' \neq s$.*

Given two partial assignments α_{i-1} and α_i in a trace such that $\alpha_{i-1} \xrightarrow{t:s} \alpha_i$, it follows that

$$\begin{aligned} V_s^{\alpha_i,\alpha} &\leq V_s^{\alpha_{i-1},\alpha} + w_{\text{rem}}^\alpha \\ V_s^{\alpha_i,\alpha} &= V_{s'}^{\alpha_{i-1},\alpha} \text{ for all } s' \neq s \end{aligned}$$

The following lemma is immediate.

Lemma 2.16 *Let $u = u^{\alpha^\tau}$ and $\alpha^\tau = \alpha_0^\tau \subseteq \alpha_1^\tau \subseteq \cdots \subseteq \alpha_u^\tau$ be a sequence of partial assignments generated by bal-assign at iteration τ. This sequence is a trace that ends in a complete assignment $B^\tau = \alpha_u^\tau$.*

2.5.2 An Upper Bound on Approximation Gap

It is obvious that Algorithm 2 is optimal for $n = 1$ since only one assignment is possible. Now we show that for $n \geq 2$, Algorithm 2 computes, in polynomial time, assignments that are optimal to within an additive constant. The result is formally stated as Theorem 2.17.

Theorem 2.17 *Let $n \geq 2$. Given an HMR-system with m tasks and n servers, Algorithm 2 computes an assignment A in time $O(m^2 n)$ such that $L^A \leq L^O + \left(1 - \frac{1}{n-1}\right) \cdot w^O_{\text{rem}}$.*

Lemma 2.18 *Algorithm 2 runs in time $O(m^2 n)$.*

Proof. By Fact 2.14, we know that the Ford-Fulkerson algorithm takes time $O(|E| \cdot |\Delta_f|)$ to augment the network flow by $|\Delta_f|$. At iteration $\tau = 1$, max-cover takes time $O(|E| \cdot |f_1|)$, where $|f_1| \leq n$. Then at iteration $\tau = 2$, max-cover takes time $O(|E| \cdot (|f_2| - |f_1|))$, where $|f_2| \leq 2n$. The same process is repeated until $|f_m| = m$. The total running time of max-cover for all iterations thus adds up to $O(|E| \cdot (|f_1| + |f_2| - |f_1| + |f_3| - |f_2| + \cdots + |f_m|)) = O(|E| \cdot |f_m|) = O(|E| \cdot m) = O(m^2 n)$.

We implement the greedy algorithm in the bal-assign phase with a priority queue. Since there are n servers, each operation of the priority queue takes $O(\log n)$ time. During the bal-assign phase at each iteration, at most m tasks need to be assigned. This takes time $O(m \log n)$. The total running time of bal-assign for all iterations is thus $O(m^2 \log n)$.

Combining the running time of the two phases for all iterations gives time complexity $O(m^2 n)$. □

Lemma 2.18 suggests the max-cover phase is the main contributor to the time complexity of Algorithm 2. However, in a typical Hadoop system, the number of replicas for each data block is a small constant, say 2 or 3. Then the degree of each $t \in G$ is bounded by this constant. In this case, the placement graph G is sparse and $|E| = O(m + n)$.

As a result, max-cover runs in time $O(m(m+n))$. Therefore the bal-assign phase might become the main contributor to the time complexity.

Properties of optimal assignments

In order to prove the approximation bound, we first establish some properties of optimal assignments.

Definition 2.15 *Given an HMR-system, let $\mathcal{O}$ be the set of all optimal assignments, i.e., those that minimize the maximum load. Let $r_{\min} = \min\{r^A \mid A \in \mathcal{O}\}$ and let $\mathcal{O}_1 = \{O \in \mathcal{O} \mid r^O = r_{\min}\}$.*

Lemma 2.19 *Let $O \in \mathcal{O}_1$. If $\ell_s^O = k^O$, then $r_s^O = 0$ and $L_s^O = K^O$.*

Proof. Let $\ell_s^O = k^O$ for some server s. Assume to the contrary that $r_s^O \geq 1$. Then $L_s^O \geq K^O + w_{\text{rem}}^O$. Let t be a remote task assigned to s by O. By definition 2.3, $\rho(t, s')$ holds for at least one server $s' \neq s$.

Case 1: s' has at least one remote task t'. Then move t' to s and t to s'. This results in another assignment B. B is still optimal because $L_s^B \leq L_s^O$, $L_{s'}^B \leq L_{s'}^O$, and $L_{s''}^B = L_{s''}^O$ for any other server $s'' \in S - \{s, s'\}$.

Case 2: s' has only local tasks. By the definition of k^O, s' has at most k^O local tasks assigned by O. Then move t to s'. This results in another assignment B. B is still optimal because $L_s^B < L_s^O$, $L_{s'}^B = K^O + w_{\text{loc}} \leq K^O + w_{\text{rem}} \leq L_s^O$, and $L_{s''}^B = L_{s''}^O$ for any other server $s'' \in S - \{s, s'\}$.

In either case, we have shown the new assignment is in $\mathcal{O}$. However, since t becomes local in the new assignment, fewer remote tasks are assigned than in O. This contradicts that $O \in \mathcal{O}_1$. Thus, s is assigned no remote tasks, so $L_s^O = k^O w_{\text{loc}} = K^O$. □

Definition 2.16 *Let $O \in \mathcal{O}_1$. Define $M^O = \frac{H^O - K^O}{n-1}$.*

Lemma 2.20 $L^O \geq M^O$.

Proof. Let s_1 be a server of maximal local load in O, so $k^O_{s_1} = k^O$. Let $S_2 = S - \{s_1\}$. By Lemma 2.19, $L^O_{s_1} = K^O$. The total load on S_2 is $\sum_{s \in S_2} L^O_s = H^O - K^O$, so the average load on S_2 is M^O. Hence, $L^O \geq \max_{s \in S_2} L^O_s \geq \text{avg}_{s \in S_2} L^O_s = M^O$. □

Analysis of the algorithm

Assume throughout this section that $O \in \mathcal{O}_1$ and $\alpha = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_u = B$ is a trace generated by iteration $\tau = k^O$ of the algorithm. Virtual loads are all based on α, so we generally omit explicit mention of α in the superscripts of v and V.

Lemma 2.21 $w_{\text{loc}} \leq w^B_{\text{rem}} \leq w^\alpha_{\text{rem}} \leq w^O_{\text{rem}}$.

Proof. $w_{\text{loc}} \leq w^B_{\text{rem}}$ follows from the definition of a Hadoop cost function. Because $B \supseteq \alpha$, $r^B \leq r^\alpha + u^\alpha$. By Lemma 2.15, $\ell^\alpha \geq \ell^O$, so $r^\alpha + u^\alpha = m - \ell^\alpha \leq m - \ell^O = r^O$. Hence, $w_{\text{rem}}(r^B) \leq w_{\text{rem}}(r^\alpha + u^\alpha) \leq w_{\text{rem}}(r^O)$ by monotonicity of $w_{\text{rem}}(\cdot)$. It follows by definition of the w^β_{rem} notation that $w^B_{\text{rem}} \leq w^\alpha_{\text{rem}} \leq w^O_{\text{rem}}$. □

Lemma 2.22 $k^\alpha = k^O$.

Proof. $k^\alpha \leq k^O$ because no server is assigned more than τ local tasks by max-cover at iteration $\tau = k^O$. For sake of contradiction, assume $k^\alpha < k^O$. Then $u^\alpha > 0$, because otherwise $\alpha = B$ and $L^B = k^\alpha \cdot w_{\text{loc}} < K^O \leq L^O$, violating the optimality of O. Let t be an unassigned task in α. By definition, $\rho(t, s)$ holds for some server s. Assign t to s in α to obtain a new partial assignment β. We have $k^\beta \leq k^\alpha + 1 \leq k^O = \tau$. By Lemma 2.15, $\ell^\alpha \geq \ell^\beta$, contradicting the fact that $\ell^\beta = \ell^\alpha + 1$. We conclude that $k^\alpha = k^O$. □

Lemma 2.23 $L^B \leq V^B$.

Proof. By definition, $L_s^B = \sum_{t:B(t)=s} w(t,B)$ and $V_s^B = \sum_{t:B(t)=s} v(t,B)$. By Lemma 2.21, $w_{\text{rem}}^B \le w_{\text{rem}}^\alpha$, and thus $w(t,B) \le v(t,B)$. It follows that $\forall s \in S$, $L_s^B \le V_s^B$. Therefore $L^B \le V^B$, because $L^B = \max_s L_s^B$ and $V^B = V^B = \max_s V_s^B$. □

For the remainder of this section, let s_1 be a server such that $\ell_{s_1}^\alpha = k^O$. Such a server exists by Lemma 2.22. Let $S_2 = S - \{s_1\}$ be the set of remaining servers. For a partial assignment $\beta \supseteq \alpha$, define N^β to be the average virtual load under β of the servers in S_2. Formally,

$$N^\beta = \frac{\sum_{s \in S_2} V_s^\beta}{|S_2|} = \frac{\ell^\beta w_{\text{loc}} + r^\beta w_{\text{rem}}^\alpha - V_{s_1}^\beta}{n-1}$$

To obtain the approximation bound, we compare N^β with the similar quantity M^O for the optimal assignment. For convenience, we let $\delta = w_{\text{rem}}^O/(n-1)$.

Lemma 2.24 *Let* $\beta = \alpha_i \xrightarrow{t:s} \alpha_{i+1} = \beta'$. *Then*

$$V_s^\beta \le N^\beta \le M^O - \delta.$$

Proof. Proof by a counting argument. By Lemma 2.22, we have $k^\alpha = k^O$, so $\ell_{s_1}^\beta \ge \ell_{s_1}^\alpha = k^O$. Hence, $V_{s_1}^\beta \ge K^O$. By Lemma 2.15, we have $\ell^\alpha \ge \ell^O$. Let $d = \ell^\beta - \ell^O$. $d \ge 0$ because $\ell^\beta \ge \ell^\alpha \ge \ell^O$. Because $\ell^\beta + r^\beta + u^\beta = m = \ell^O + r^O$, we have $r^\beta + u^\beta + d = r^O$. Also, $u^\beta \ge 1$ since t is unassigned in β. Then by Lemma 2.21,

$$\begin{aligned}
(n-1)N^\beta &= \ell^\beta w_{\text{loc}} + r^\beta w_{\text{rem}}^\alpha - V_{s_1}^\beta \\
&= (\ell^O + d) w_{\text{loc}} + (r^O - u^\beta - d) w_{\text{rem}}^\alpha - V_{s_1}^\beta \\
&\le \ell^O w_{\text{loc}} + (r^O - u^\beta) w_{\text{rem}}^O - K^O \\
&\le (n-1)M^O - w_{\text{rem}}^O.
\end{aligned}$$

Hence, $N^\beta \le M^O - \delta$.

Now, since β is part of a trace, we have $V_s^\beta \leq V_{s'}^\beta$ for all $s' \in S$. In particular, $V_s^\beta \leq N^\beta$, since N^β is the average virtual load of all servers in S_2. We conclude that $V_s^\beta \leq N^\beta \leq M^O - \delta$. □

Proof of Theorem 2.17: Lemma 2.18 shows that the time complexity of Algorithm 2 is $O(m^2 n)$. Now we finish the proof for the approximation bound.

Let s be a server of maximum virtual load in B, so $V_s^B = V^B$. Let i be the smallest integer such that $\alpha_i|_s = B|_s$, that is, no more tasks are assigned to s in the subtrace beginning with α_i.

Case 1: $i = 0$: Then $\ell_s^{\alpha_0} \leq k^{\alpha_0} = k^O$ by Lemma 2.22, and $r^{\alpha_0} = 0$, so $V^B = V_s^{\alpha_0} \leq K^O$. Hence, $V^B \leq K^O \leq L^O$.

Case 2: $i > 0$: Then $\beta = \alpha_{i-1} \xrightarrow{t:s} \alpha_i = \beta'$ for some task t. By lemma 2.24, $V_s^\beta \leq M^O - \delta$, so using Lemma 2.21,

$$V_s^{\beta'} \leq V_s^\beta + w_{\text{rem}}^\alpha \leq M^O - \delta + w_{\text{rem}}^O.$$

Then by Lemma 2.20,

$$V^B = V_s^B = V_s^{\beta'} \leq M^O + w_{\text{rem}}^O - \delta \leq L^O + w_{\text{rem}}^O - \delta.$$

Both cases imply that $V^B \leq L^O + w_{\text{rem}}^O - \delta$. By Lemma 2.23, we have $L^B \leq V^B$. Because the algorithm chooses an assignment with least maximum load as the output A, we have $L^A \leq L^B$. Hence,

$$L^A \leq L^O + w_{\text{rem}}^O - \delta \; = \; L^O + \left(1 - \frac{1}{n-1}\right) \cdot w_{\text{rem}}^O$$

□

2.5.3 A Matching Lower Bound

In this section, we show a matching lower bound for Algorithm 2 on a restricted class of UHMR-systems. The result is proved through a series of lemmas.

Theorem 2.25 *Let $n \geq 3$ and O be an assignment that minimizes L^O. There exists a UHMR-system and an assignment A, such that A is a possible output of Algorithm 2 and $L^A = L^O + \left(1 - \frac{1}{n-1}\right) w_{\text{rem}}$, where w_{rem} is the cost for any remote task.*

Assume m tasks in T and n servers in S. Let $m = 2n^2 - 3n + 2$. We construct a placement graph G_ρ that consists of two disjoint complete bipartite subgraphs $G_1 = ((T_1, S_1), E_1)$ and $G_2 = ((T_2, S_2), E_2)$, such that $|T_1| = 2n^2 - 5n + 4$, $|T_2| = 2n - 2$, $|S_1| = 1$, and $|S_2| = n - 1$. Let $S_1 = \{s_1\}$ and $S_2 = \{s_2, s_3, \ldots, s_n\}$. There exists an edge $(t, s) \in E_i$ for all $t \in T_i$, $s \in S_i$, and $i \in \{1, 2\}$. G_ρ is shown in Figure 2.4.

Let w_{loc} be any positive rational number. Let

$$w_{\text{rem}} = (2n - 2)/(2n - 3) \cdot w_{\text{loc}} \tag{2.3}$$

Let $w(\cdot)$ be the uniform Hadoop cost function with parameters ρ, w_{loc}, and $w_{\text{rem}}(\cdot) = w_{\text{rem}}$. As a result, a UHMR-system $\mathcal{U}$ is defined by $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$. In the following discussion, we study the task assignments for this UHMR-system $\mathcal{U}$.

We first describe an assignment O and show it is optimal and thus minimizes L^O. O assigns $2n - 2$ tasks in T_1 to server s_1 as local tasks and all tasks in T_2 to server s_2 as local tasks. In addition, O evenly assigns the remaining $2n^2 - 7n + 6$ tasks in T_1 to servers in $S_2 - \{s_2\}$ as remote tasks. Therefore under O, s_1 is assigned $2n - 2$ local tasks, s_2 is assigned $2n - 2$ local tasks, and each server in $S_2 - \{s_2\}$ is assigned $(2n^2 - 7n + 6)/(n - 2) = 2n - 3$ remote tasks. O is described in Figure 2.5. Each column corresponds to a server. Rectangles in each column represent the tasks assigned to

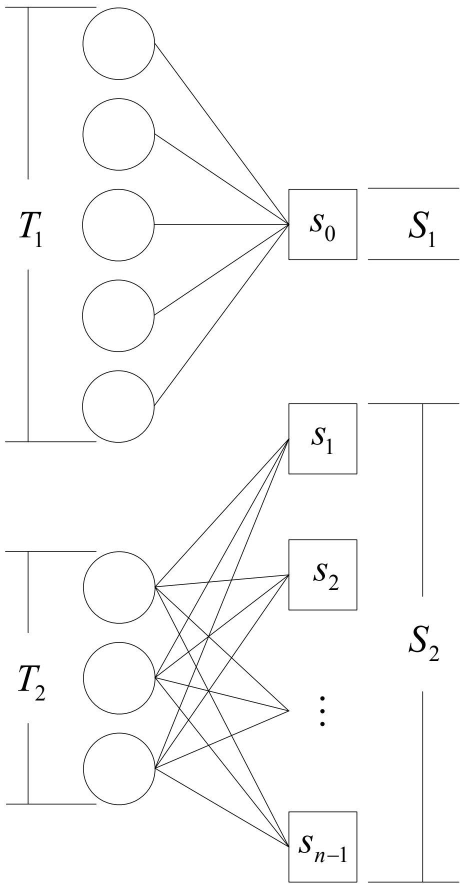

Figure 2.4: The placement graph.

this server. Tasks are distinguished by their colors. A local task is represented by a white rectangle, while a remote task is represented by a gray rectangle. The height of a column represents the server load.

Lemma 2.26 *Assignment O minimizes L^O and $L^O = (2n-2)w_{\text{loc}} = (2n-3)w_{\text{rem}}$.*

Proof. Since $n \geq 3$, we have $2n-2 > 2n-3 > 0$. By (2.3), we have $(2n-2)w_{\text{loc}} = (2n-3)w_{\text{rem}}$. Thus all servers have equal load under O and it follows that

$$L^O = (2n-2)w_{\text{loc}} = (2n-3)w_{\text{rem}} \quad (2.4)$$

$$H^O = nL^O \quad (2.5)$$

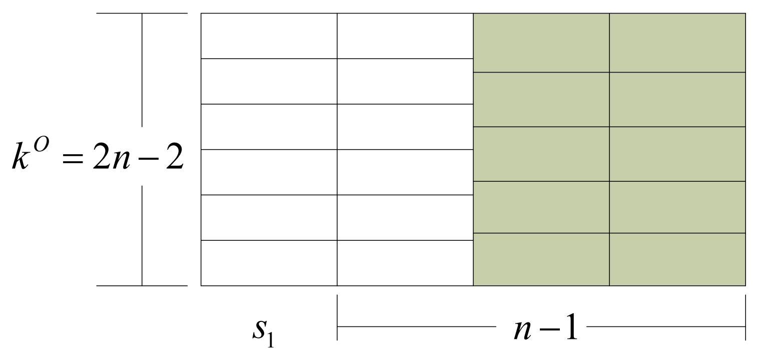

Figure 2.5: The assignment O.

Now we show that O is optimal. For the sake of contradiction, assume there is another assignment O' such that $L^{O'} < L^O$. It immediately follows that $H^{O'} \leq nL^{O'}$ because $L^{O'}$ is the maximum load under O'. It then follows from (2.5) that

$$H^{O'} \leq nL^{O'} < nL^O = H^O$$

This implies that O' assigns more local tasks than O, otherwise it cannot produce less total load. Because O assigns $2(2n-2)$ local tasks in total, O' must have assigned at least $2(2n-2)+1 = 4n-3$ local tasks. Since $|T_2| = 2n-2$, O' assigns at least $2n-1$ tasks in T_1 as local tasks and they can only be local on s_1. Therefore, this gives

$$L^{O'} \geq L_{s_1}^{O'} \geq (2n-1)w_{\text{loc}} \tag{2.6}$$

Combining (2.4) and (2.6) gives

$$L^{O'} \geq (2n-1)w_{\text{loc}} > (2n-2)w_{\text{loc}} = L^O$$

which contradicts the assumption that $L^{O'} < L^O$. Thus O is optimal. □

We then describe an assignment $A^\star$. $A^\star$ assigns $2n-2$ tasks in T_1 to s_1 as local tasks and all the $2n-2$ tasks in T_2 evenly to servers in S_2 as local tasks. In addition, $A^\star$ assigns $2n-4$ tasks in T_1 to s_2 as remote tasks and $2n-5$ tasks in T_1 to each server in $S_2 - \{s_2\}$ as remote tasks. As a result, s_1 is assigned $2n-2$ local tasks, s_2 is assigned 2 local tasks and $2n-4$ remote tasks, and each server in $S_2 - \{s_2\}$ is assigned 2 local tasks and $2n-5$ remote tasks. $A^\star$ is described in Figure 2.6.

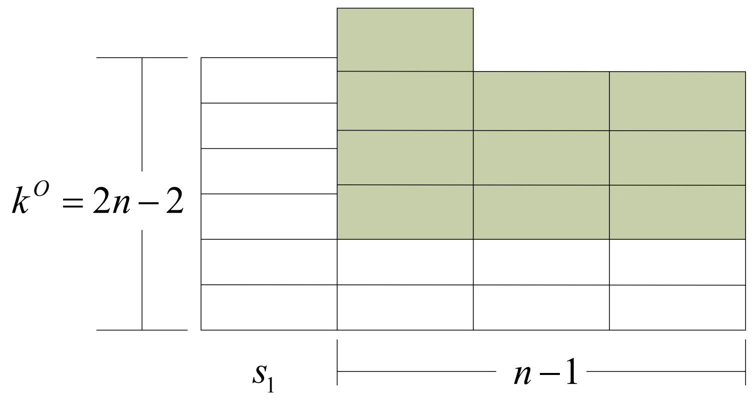

Figure 2.6: The assignment $A^\star$.

Lemma 2.27 $L^{A^\star} = L^O + \left(1 - \frac{1}{n-1}\right) w_{\text{rem}}$.

Proof. Server s_1 is assigned $2n-2$ local tasks by $A^\star$ and thus

$$L_{s_1}^{A^\star} = (2n-2)w_{\text{loc}} \tag{2.7}$$

It is straightforward that s_2 has the largest load among servers in S_2, because s_2 is assigned one more remote task than any other server in S_2. We have

$$L_{s_2}^{A^\star} = 2w_{\text{loc}} + (2n-4)w_{\text{rem}} \tag{2.8}$$

Since $w_{\text{rem}} > w_{\text{loc}}$, comparing (2.7) and (2.8) gives

$$L_{s_2}^{A^\star} > L_{s_1}^{A^\star}$$

Thus, server s_2 has the largest load under $A^\star$, i.e., $L^{A^\star} = L_{s_2}^{A^\star}$. Therefore, by (2.3) and (2.8) we have

$$L^{A^\star} = \frac{2n^2 - 4n + 1}{n - 1} w_{\text{rem}} \tag{2.9}$$

Combining (2.4) and (2.9) gives

$$L^{A^\star} - L^O = \frac{2n^2 - 4n + 1}{n - 1} w_{\text{rem}} - (2n - 3) w_{\text{rem}} = \frac{n - 2}{n - 1} w_{\text{rem}} \tag{2.10}$$

Hence,

$$L^{A^\star} = L^O + \left(1 - \frac{1}{n - 1}\right) w_{\text{rem}} \tag{2.11}$$

□

Definition 2.17 *Given a UHMR-system, let $s \in S$ be a server and A be an assignment. The* load profile *P_s^A of s under A is an ordered pair (x, y), where x is the number of local tasks and y the number of remote tasks assigned to s by A.*

Definition 2.18 *Given a UHMR-system, two assignments A and B are* isomorphic *(denoted $A \cong B$) iff there exists a permutation $\mathcal{P} : S \to S$, such that $P_s^A = P_{s'}^B$ for all $s \in S$, where $s' = \mathcal{P}(s)$. The permutation $\mathcal{P}$ is called a* witness *of the isomorphism between A and B.*

It follows immediately that if $A \cong B$, then $L^A = L^B$.

By definition, for the given UHMR-system $\mathcal{U}$, $P_{s_1}^{A^\star} = (2n - 2, 0)$, $P_{s_2}^{A^\star} = (2, 2n - 4)$, and $P_s^{A^\star} = (2, 2n - 5)$ for all $s \in S_2 - \{s_2\}$. We show that there exists an execution of Algorithm 2 such that it computes an assignment A that is isomorphic to $A^\star$, and thus $L^A = L^{A^\star}$.

Algorithm 2 does not specify the order of choosing tasks in both max-cover and bal-assign, nor does it specify the tie-breaking orders in bal-assign when more than one server has minimal virtual load. Therefore, there are many possible executions. We describe an execution and prove its various properties.

Recall that for any $1 \le \tau \le m$, we use the notation $\alpha^\tau = \alpha_0^\tau$ to be the partial assignment computed by the max-cover phase and B^τ the assignment computed by the bal-assign phase, in the τ-th iteration. The important part is the execution of the first iteration $\tau = 1$. It is straightforward that α^1 assigns each server 1 local task in the bal-assign phase. Let $t_1 \in T_1$ and $t_2, t_3, \ldots, t_n \in T_2$. Define $\alpha^1 = \alpha_0^1 \to \alpha_1^1 \to \cdots \to \alpha_{m-n}^1 = B^1$ as the trace in the bal-assign phase, such that $\alpha_{i-1}^1 \xrightarrow{t_i:s_i} \alpha_i^1$ for $1 \le i \le n$. Hence, in the partial assignment α_n^1, each server is assigned exactly 2 local tasks. For the remaining iterations $2 \le \tau \le m$, the order of choosing tasks and servers is arbitrary.

Lemma 2.28 *For any $1 \le \tau \le m$, assignment B^τ assigns 2 local tasks to any server $s \in S_2$.*

Proof. For $\tau = 1$, α_n^1 assigns all the $2n-2$ tasks in T_2 evenly to the $n-1$ servers in S_2. Because all the unassigned tasks in α_n^1 are in T_1 and they cannot be local on S_2 servers, B^1 assigns 2 local tasks to any server in S_2.

For $\tau = 2$, α^2 is computed by max-cover with threshold 2. The $n-1$ servers in S_2 only accept the $2n-2$ tasks in T_2 as local tasks. The only pattern that maximizes the flow size when $\tau = 2$ is to evenly distribute T_2 tasks to S_2 servers. Therefore, α^2 assigns each S_2 server 2 local tasks from T_2. Similarly, because all the unassigned tasks in α^2 are in T_1 and they cannot be local on S_2 servers, B^2 assigns 2 local tasks to any server in S_2.

Now we consider iteration $\tau = 3$. Max-cover takes the partial assignment α^2 as input and augments it into α^3 by looking for augmenting paths in the residual graph. Figure 2.7 describes the lower half of the residual graph for α^2. Each circle represents a task in T_2 and each rectangle represents a server in S_2. Each directed edge between v and a server has capacity 2. All the other directed edges have capacity 1.

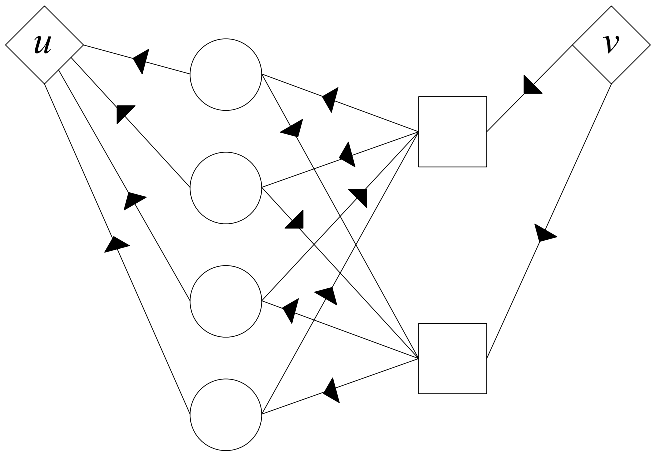

Figure 2.7: The lower half of the residual graph for $\alpha(2)$.

When τ is increased from 2 to 3, we add a directed edge between each server and v with capacity 1 in the residual graph. However, since there are neither directed edges from u to any $t \in T_2$ (all T_2 tasks have already been assigned), nor directed edges from $t \in T_1$ to servers in S_2 (S_2 servers do not accept T_1 tasks as local ones), there do not exist any augmenting paths from u to v through S_2 servers. Therefore, the 2 local tasks assigned to each S_2 server by α^2 remain the same in α^3. Again, because all the unassigned tasks in α^3 are in T_1 and they cannot be local on S_2 servers, B^3 assigns 2 local tasks to any server in S_2.

The claim is true for all B^τ with $\tau > 3$ due to the same reasoning, and thus the proof is complete. □

Lemma 2.29 *For any* $1 \leq \tau \leq 2n - 2$, *we have* $L^{B^\tau} > \tau \cdot w_{\text{loc}}$.

Proof. Since $\tau \leq 2n-2 < |T_1|$, server s_1 is assigned τ local tasks by α^τ in the max-cover phase. Following Lemma 2.28, all T_2 tasks are assigned to S_2 servers. Therefore, all the tasks assigned to server s_1 in bal-assign phase are from T_1 and thus are local. Let $P_{s_1}^{B^\tau} = (k, 0)$.

Case 1: Assume $k > \tau$, then $L^{B^\tau} \geq L_{s_1}^{B^\tau} = k \cdot w_{\text{loc}} > \tau \cdot w_{\text{loc}}$.

Case 2: Assume $k = \tau$, then all the remaining $|T_1| - \tau$ tasks in T_1 are assigned to S_2 servers as remote tasks. Moreover, by Lemma 2.28, each server in S_2 is assigned 2 local tasks. Then by comparing the average load M^{B^τ} of S_2 servers with $\tau \cdot w_{\text{loc}}$, we have

$$\begin{aligned}
& M^{B^\tau} - \tau \cdot w_{\text{loc}} \\
= & \left(2 \cdot w_{\text{loc}} + \frac{|T_1| - \tau}{n-1} \cdot w_{\text{rem}}\right) - (\tau \cdot w_{\text{loc}}) \\
= & \frac{2n^2 - 5n + 4 - \tau}{n-1} w_{\text{rem}} - (\tau - 2) w_{\text{loc}} \\
\geq & \frac{2n^2 - 5n + 4 - 2n + 2}{n-1} w_{\text{rem}} - (2n - 2 - 2) w_{\text{loc}} \\
= & \ 0
\end{aligned} \tag{2.12}$$

When $\tau < 2n-2$, (2.12) holds as a strict inequality. In this case, we have

$$L^{B^\tau} \geq M^{B^\tau} > \tau \cdot w_{\text{loc}}$$

When $\tau = 2n-2$, (2.12) holds as an equality. In this case, since $(n-1) \nmid (2n^2 - 7n + 6)$ and a single task is not divisible, the maximum load is strictly greater than the average load. Thus,

$$L^{B^\tau} > M^{B^\tau} = \tau \cdot w_{\text{loc}}$$

This completes the proof. □

Lemma 2.30 *For any $s, s' \in S$ and $1 \leq \tau \leq 2n-2$, we have $|L_s^{B^\tau} - L_{s'}^{B^\tau}| \leq w_{\text{rem}}$.*

Proof. Assume servers p and q have maximum and minimal load under B^τ, respectively. Let i be the smallest integer such that $\alpha_i^\tau|p = B^\tau|p$. Following Lemma 2.29, $L_p^{B^\tau} > \tau \cdot w_{\text{loc}}$ and thus p is assigned at least 1 task in the bal-assign phase, which indicates that $i > 0$. Then $\beta = \alpha_{i-1}^\tau \xrightarrow{t:p} \alpha_i^\tau = \beta'$ for some task t. Since β and β' are in a trace, we have

$$V_p^{\beta'} - V_q^{\beta'} \leq w(t, \beta') \leq w_{\text{rem}}$$

Because $\beta'|p = B^\tau|p$ and $\beta'|q \subseteq B^\tau|q$, we have

$$V_p^{B^\tau} - V_q^{B^\tau} \leq V_p^{\beta'} - V_q^{\beta'} \leq w_{\text{rem}} \tag{2.13}$$

For any $s, s' \in S$, we have

$$|V_s^{B^\tau} - V_{s'}^{B^\tau}| \leq V_p^{B^\tau} - V_q^{B^\tau} \tag{2.14}$$

Combing (2.13) and (2.14) gives

$$|V_s^{B^\tau} - V_{s'}^{B^\tau}| \leq w_{\text{rem}}$$

As the HMR-system $\mathcal{U}$ is uniform, w_{rem} is a constant and thus $L_s^{B^\tau} = V_s^{B^\tau}$ for any $s \in S$. Therefore,

$$|L_s^{B^\tau} - L_{s'}^{B^\tau}| \leq w_{\text{rem}}$$

□

Lemma 2.31 *For any* $1 \leq \tau \leq 2n-2$, *we have* $P_{s_1}^{B^\tau} = (2n-2, 0)$.

Proof. Lemma 2.28 indicates that all S_2 tasks are assigned to T_2 servers. Therefore, s_1 is assigned only T_1 tasks. Let $P_{s_1}^{B^\tau} = (k, 0)$.

Case 1: Assume $k \leq 2n-3$. Then there are at least $|T_1| - (2n-3) = 2n^2 - 7n + 3$ tasks in T_1 are assigned to S_2 servers as remote tasks. Since we have

$$\left\lceil \frac{2n^2 - 7n + 3}{n-1} \right\rceil = 2n - 4,$$

there is one server $s \in S_2$ that is assigned at least $2n-4$ remote tasks. Thus,

$$\begin{aligned} L_s^{B^\tau} - L_{s_1}^{B^\tau} &\geq [2w_{\text{loc}} + (2n-4)w_{\text{rem}}] - (2n-3)w_{\text{loc}} \\ &= w_{\text{rem}} + (2n-5)(w_{\text{rem}} - w_{\text{loc}}) > w_{\text{rem}}, \end{aligned}$$

which violates Lemma 2.30.

Case 2: Assume $k \geq 2n-1$. Then there are at most $|T_1| - (2n-1) = 2n^2 - 7n + 5$ tasks in T_1 are assigned to T_2 servers as remote tasks. Since we have

$$\left\lfloor \frac{2n^2 - 7n + 5}{n-1} \right\rfloor = 2n - 5,$$

there is one server $s \in S_2$ that is assigned at most $2n-5$ remote tasks. Thus,

$$\begin{aligned} L_{s_1}^{B^\tau} - L_s^{B^\tau} &\geq (2n-1)w_{\text{loc}} - [2w_{\text{loc}} + (2n-5)w_{\text{rem}}] \\ &= (2n-3)w_{\text{loc}} - (2n-5)w_{\text{rem}} \\ &= (2n-3)\frac{2n-3}{2n-2}w_{\text{rem}} - (2n-5)w_{\text{rem}} \\ &= \frac{2n-1}{2n-2}w_{\text{rem}} > w_{\text{rem}}, \end{aligned}$$

which violates Lemma 2.30.

Therefore we conclude that $k = 2n-2$ and thus $P_{s_1}^{B^\tau} = (2n-2, 0)$. □

Lemma 2.32 *For any $s \in S_2$ and $1 \leq \tau \leq 2n-2$, we have $P_s^{B^\tau} \in \{(2, 2n-5), (2, 2n-$*

$4)\}$.

Proof. By Lemma 2.28, any server $s \in S_2$ is assigned 2 local tasks. Let $P_s^{B^\tau} = (2, y)$. By Lemmas 2.30 and 2.31, we have

$$|L_s^{B^\tau} - L_{s_1}^{B^\tau}| \;=\; |(2w_{\text{loc}} + y \cdot w_{\text{rem}}) - (2n-2)w_{\text{loc}}| \leq w_{\text{rem}} \tag{2.15}$$

Solving (2.15) for y gives

$$\begin{aligned} -w_{\text{rem}} \leq (2w_{\text{loc}} + y \cdot w_{\text{rem}}) - (2n-2)w_{\text{loc}} \leq w_{\text{rem}} \\ -w_{\text{rem}} \leq (y \cdot w_{\text{rem}}) - (2n-4)\frac{2n-3}{2n-2}w_{\text{rem}} \leq w_{\text{rem}} \\ 2n - 6 + \frac{1}{n-1} \leq y \leq 2n - 4 + \frac{1}{n-1} \end{aligned} \tag{2.16}$$

The integer solution to (2.16) is in $\{2n-5, 2n-4\}$. This completes the proof. $\square$

Lemma 2.33 *For $1 \leq \tau \leq 2n-2$, we have $B^\tau \cong A^\star$.*

Proof. By Lemmas 2.31 and 2.32, we have already known the range of load profiles. Now we identify the exact number of servers that have each load profile. Assume there are x servers in S_2 that have profile $(2, 2n-4)$, and the remaining $n-1-x$ ones have profile $(2, 2n-5)$. Counting the total number of T_1 tasks gives

$$2n - 2 + (2n-4)x + (2n-5)(n-1-x) = |T_1| = 2n^2 - 5n + 4$$

Solving for x gives $x = 1$. This means that there is only 1 server that has load profile $(2, 2n-4)$. Denote this server by q. We construct a permutation $\mathcal{P} : S \to S$, such that $\mathcal{P}(s_1) = s_1$ and $\mathcal{P}(q) = s_2$. Then $\mathcal{P}$ is a witness that $B^\tau \cong A^\star$. $\square$

Lemma 2.34 *For any $2n-1 \leq \tau \leq m$, we have $L^{B^\tau} > L^{A^\star}$.*

Proof. After the max-cover phase, s_1 is assigned at least $2n-1$ local tasks, and thus

$$L^{B^\tau} \geq L^{\alpha^\tau} \geq (2n-1)w_{\text{loc}} \tag{2.17}$$

Combining (2.9) and (2.17) gives

$$\begin{aligned} L^{B^\tau} - L^{A^\star} &\geq (2n-1)w_{\text{loc}} - \frac{2n^2-4n+1}{n-1}w_{\text{rem}} \\ &= (2n-1)\frac{2n-3}{2n-2}w_{\text{rem}} - \frac{2n^2-4n+1}{n-1}w_{\text{rem}} \\ &= \frac{1}{2n-2}w_{\text{rem}} > 0 \end{aligned} \tag{2.18}$$

Therefore, for any $2n-1 \leq \tau \leq m$, we have $L^{B^\tau} > L^{A^\star}$. □

Finally we prove the theorem.

Proof of Theorem 2.25: By Lemmas 2.33 and 2.34, we conclude that Algorithm 2 chooses an assignment $A \in \{B^\tau | 1 \leq \tau \leq 2n-2\}$ and $A \cong A^\star$. This fact and Lemma 2.27 together show that $L^A = L^{A^\star} = L^O + \left(1 - \frac{1}{n-1}\right) w_{\text{rem}}$. This completes the proof. □

2.6 Further Discussions

We showed that the simple round robin algorithm runs in linear time but computes assignments that could deviate from the optimum by a multiplicative factor $(w_{\text{rem}}/w_{\text{loc}})$. The remote cost w_{rem} is usually considerably higher than the local cost w_{loc}, and thus the performance of the round robin algorithm can be a problem. We exploited network flow to achieve a better approximation. FLOWT produces assignments that are optimal to within an additive gap. However, this algorithm has a worse time complexity.

There are several potential improvements we can explore. We discussed the implementation of FLOWT based on the Ford-Fulkerson augmenting path algorithm [30], which is the major contributor to the complexity of the algorithm. A possible refinement is to use approximated maximum flow algorithms that run faster [16]. The effects of approximated maximum flows on the approximation bound of task assignments need to be analyzed.

If some constraints on the data placement are available, one can potentially take advantage of such additional information. For example, the Hadoop Distributed File System employs some policies to balance the placement of data block replicas [9]. When replicas are evenly distributed in the cluster, following Hall's theorem [44], one can easily show that all tasks can be evenly assigned to servers as local tasks. In the bipartite graph representation, duplicate each server node and create a regular bipartite graph. Then an optimal assignment of tasks corresponds to a perfect matching in this graph. A perfect matching in regular bipartite graphs can be found in time linear to the number of edges [17, 32], and even in sublinear time with additional requirement on nodes' degree [37].

Chapter 3

Competitive Analysis of Online Assignments

We can only see a short distance ahead, but we can see plenty there that needs to be done.

– Alan Turing

3.1 Introduction

3.1.1 Scheduling Under the Restricted Assignment Model

In Chapter 2, we presented an abstract model for Hadoop. We showed that the Hadoop task assignment (HTA) problem is $\mathcal{NP}$-hard. We also analyzed two approximation algorithms for this problem. In this chapter, we connect the HTA problem to the minimum makespan scheduling problem under the restricted assignment model [6]. In the restricted assignment model, each task is associated with a set of feasible servers, which are similar to the set of local servers under the Hadoop model. However, the restricted assignment model requires that tasks only be assigned to feasible servers. Another distinction is that the Hadoop model assumes identical cost for all tasks assigned to local servers, while the restricted assignment model allows different costs for different tasks.

Most of this chapter focuses on online assignments. When tasks come sequentially in an online fashion, an online algorithm needs to choose which server to assign the task to upon its arrival, without the knowledge about the future. A natural algorithm, GREEDY, works by assigning each task to a feasible server of minimal load. Ties are broken arbitrarily. GREEDY is a useful building block in many algorithm designs and has been proven to work pretty well under different settings [6, 12, 29, 39, 40].

A popular performance measure for online algorithms is called the *competitive ratio* [71]. Consider an optimal offline algorithm OPT that knows the sequence of tasks in advance and has unbounded computational power. Given a scheduling problem instance, let A be an assignment computed by GREEDY and O be an assignment computed by OPT, and let L^A and L^O be the respective makespans. The competitive ratio of GREEDY is defined as the maximum of L^A/L^O over all possible inputs. This measure can be regarded as the worst case penalty of operating online.

3.1.2 Witness Graphs and Token Production Systems

The main result on the competitive ratio bound for GREEDY in this chapter is based on the analysis of the lower bound on total task load. Two novel structures called *witness graphs* and *token production systems (TPSs)* are developed for this purpose.

Witness graphs are ordered weighted multidigraphs. Each witness graph embeds two task assignments and allows them to be compared in the same framework. The servers and tasks of the scheduling problem are in one-to-one correspondence with the vertices and edges of the witness graph, respectively. The loads of a server under two assignments are then naturally represented by the weights of incoming and outgoing edges for each vertex.

Token production systems have simple algebraic structure and connect ideas from vector addition systems (VASs) [47] in computer science with input-output models [56] in economics. TPSs augment VASs with a cost function and scalar multiplication. In contrast to input-output models, TPSs are able to express sequences of actions, and thus can be used to analyze problems in a much easier way, where not only the final state but also each intermediate state is considered. A TPS defines a finite set of rational vectors and their costs. For convenience, we call each position of a vector a *bucket*, and the value at that position the number of *tokens* in that bucket. Taking negative token values as demands and positive ones as supplies, each vector defined by a TPS describes an action that relates demands and supplies of different buckets. An *activity* defines a sequence of actions in order to produce tokens in some desired way. The cost associated with each action allows one to compare the cost of different activities that result in the same token production.

3.1.3 Related Work

Under the restricted assignment model, Azar, Naor, and Rom [6] show that the best competitive ratio that can be achieved by any deterministic online algorithm is $\lceil \log{(n+1)} \rceil$, where n is the number of servers. By partitioning assignments computed by GREEDY into layered sub-assignments and analyzing the adjusted weights of unfinished tasks in each layer, they prove that GREEDY achieves competitive ratio of $\lceil \log n \rceil + 1$. By bounding the load on any prefix of most loaded servers with residual weights, Buchbinder and Naor show an interesting alternative bound on the makespan of assignments computed by GREEDY [12]. This bound can be translated into a competitive ratio bound as

$$\frac{\lambda_{(1)}}{\lambda^*_{(1)}} \leq 1 + \sum_{i=0}^{\lfloor \log n \rfloor} \frac{1}{2^i} \sum_{r=2^i+1}^{\min{(n, 2^{i+1})}} \frac{\lambda^*_{(r)}}{\lambda^*_{(1)}},$$

where $\lambda_{(1)}$ is the maximum server load achieved by GREEDY assignments and $\lambda^*_{(r)}$ is the r-th highest server load achieved by OPT assignments. This refined bound matches the original result in [6] only when the optimal assignment results in totally balanced load among all servers, and is strictly better otherwise.

In many areas of research in computer science, operations research and economics, problems arise with precedence and dependency relations. Other problems involve activities with sequential steps or progressive evolutions. Examples include but are not limited to, production processes, project management, job shop scheduling, and activity analysis. Many tools and models have been proposed to represent and analyze these problems, such as partially ordered sets, directed acyclic graphs, vector addition systems, and input-output models. These tools have been extensively studied in the literature and shown to be effective in many applications.

In computer science, vector addition systems (VASs) [47] and the equivalent Petri nets are one of the most popular formal models for the representation and analysis of parallel processes. A central problem with VASs is to decide whether a target configuration is reachable from an initial configuration [26]. Many computational problems inside and outside the parallel processes reduce to this reachability problem [42, 48]. Informally speaking, a vector addition system is a finite set $A \subseteq \mathbb{Z}^n$. Given two configurations $\boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}$, the reachability problem asks whether there exists a sequence of actions $\boldsymbol{x} \xrightarrow{\boldsymbol{a}_1} \boldsymbol{x}_1 \rightarrow \cdots \xrightarrow{\boldsymbol{a}_k} \boldsymbol{x}_k = \boldsymbol{y}$ such that $\boldsymbol{a}_i \in A$ and $\boldsymbol{x}_i \geq \boldsymbol{0}$ for all $1 \leq i \leq k$. The first partial proof was proposed by Sacerdote and Tenney [70] in 1977. Mayr [58] completed the proof in 1981, Kosaraju [50] simplified the proof in 1982, and Lambert [53] further simplified the proof in 1992. Some work extends Petri nets by the Timed Petri nets (TPNs) with clocks and real-time constraints [7, 69]. An further extension to TPNs assigns firing costs to transitions and storage costs to places [1, 2].

In economics, input-output models provide techniques for the quantitative study of economic principles [68]. Input-output models are introduced by Leontief [56] to study the interdependencies between different sectors of a country's economy or between branches of different, even competing economies. A matrix is used to represent how each sector depends on others' both as consumer of their outputs and as producer of their inputs. This quantitative economic technique is later used by Koopmans [49] in the context of activity analysis. An activity analysis is described in terms of goods and technologies. Each technology consumes a bundle of goods and produces another with some operational cost. Each technology should be operated at a non-negative level. The decision variables in an activity analysis include the level at which each technology should be operated. A general question in activity analysis asks how to operate each technology such that some quantity of goods are produced with minimum cost. Activity analysis also provides answers to many questions involving changing polices and resource constraints.

3.1.4 Our Contributions

In this chapter, we introduce a new approach for competitive analysis of online assignments.

In Section 3.2, we establish the connection between the restricted HTA problem and the minimum makespan scheduling problem under the restricted assignment model. In Section 3.4.1, to analyze the competitive ratio of GREEDY, we define a different but related problem called the *minimum total load problem*. The total load of an online scheduling system is the total run time of all tasks to be assigned. Given two parameters d and ℓ, let $\mathcal{N}(d, \ell)$ be the set of scheduling systems such that GREEDY can produce assignments with makespan d and OPT produces assignments with makespan at most ℓ. The minimum total load problem is to determine the minimum total load of any system in $\mathcal{N}(d, \ell)$. We translate a lower bound for the minimum total load problem into an upper bound for the competitive ratio of GREEDY.

In Section 3.5, we introduce witness graphs. Given any two assignments A, B in an online scheduling system σ, we define and construct witness graph $G_\sigma^{A,B}$. The *weight* of a witness graph is the total edge weight of the graph. We define a family of witness graphs $\mathcal{G}(d, \ell)$ that embed scheduling systems in $\mathcal{N}(d, \ell)$, and show that a lower bound for the minimum weight of witness graphs in $\mathcal{G}(d, \ell)$ is a lower bound for the minimum total load of scheduling systems in $\mathcal{N}(d, \ell)$.

In Section 3.6, we define token production systems. To lower-bound the minimum weight of witness graphs in $\mathcal{G}(d, \ell)$, we define a notion of *token production cost* in a family of TPSs and construct activities to embed witness graphs. We show that the cost of the activity equals the weight of the embedded witness graph, and thus the minimum production cost provides a lower bound for the minimum graph weight. We also show many properties of TPSs and present results on various reachability problems.

Finally in Section 3.7, we solve the minimum token production cost problem using a direct combinatorial analysis. This result, together with the reductions mentioned above, gives a new upper bound on the competitive ratio of GREEDY. This bound refines the original result in [6] and is incomparable to the best such result known to date [12].

3.2 Restricted Hadoop Task Assignment Problem

Definition 3.1 (Restricted Task assignment) *Let S be a set of servers, T be a set of tasks, and $\rho \subseteq T \times S$ be a data placement relation. Define a* restricted task assignment *to be a function $A : T \to S$ such that $\rho(t, A(t))$ holds for all t.*

Problem 3.1 Restricted Hadoop Task Assignment Problem (RHTA)

1. **Instance:** A HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.
2. **Objective:** Find a restricted task assignment A that minimizes L^A.

Recall that the HTA problem is shown to be $\mathcal{NP}$-hard in Chapter 2. In contrast, we show there exist efficient algorithms for the RHTA problem. In particular, ignoring the *bal-assign* phase of FLOWT gives the following algorithm that solves the RHTA problem.

Algorithm 3: A variant of FLOWT for RHTA.

```
   Input: an HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$
1  define $A$ as an assignment
2  define $\alpha$ as a partial assignment
3  $\alpha(t) \leftarrow \perp$ (task $t$ is unassigned) for all $t$
4  for $\tau \leftarrow 1$ to $m$ do
5      $\alpha \leftarrow$ max-cover$(G_\rho, \tau, \alpha)$
6      if $\alpha$ assigns all tasks in $T$ then
7          $A \leftarrow \alpha$
8          break
   Output: assignment $A$
```

Theorem 3.1 *Given an HMR-system with m tasks and n servers, Algorithm 3 computes an optimal restricted task assignment in time $O(m^2 n)$.*

Proof. Apparently the complexity of Algorithm 3 is no worse than Algorithm 2. By Lemma 2.18, Algorithm 2 runs in time $O(m^2 n)$ and so does Algorithm 3. The output A is a restricted task assignment, because max-cover phase only assign tasks to local servers.

It remains to show that Algorithm 3 computes an optimal restricted assignment. Assume for the sake of contradiction that there exist another restricted assignment O such that $L^O < L^A$. Consider the iteration when $\tau = L^O / w_{\text{loc}}$ during the execution of Algorithm 3. Because the algorithm does not terminate in this iteration, the partial assignment α^τ in this iteration does not assign all tasks in T. Therefore, for the total number of local tasks assigned under α^τ and O, we have $\ell^{\alpha^\tau} < \ell^O$. On the other hand, by Lemma 2.15, $\ell^{\alpha^\tau} \geq \ell^O$. This is a contradiction, and thus there does not exist such an assignment O. Therefore, we conclude that A is optimal. □

By restricting all tasks to be assigned locally, the RHTA problem admits efficient algorithms. If we extend the cost function such that local tasks can have different costs, the problem becomes identical to the minimum makespan scheduling problem under the restricted assignment model, which is known to be $\mathcal{NP}$-hard. In the rest of this chapter, we study the online version of this minimum makespan scheduling problem.

3.3 Online Minimum Makespan Scheduling with Restricted Assignment

In this section, we define the online minimum makespan scheduling problem under the restricted assignment model.

Given a set of tasks with their execution times and a set of servers that run in parallel, the minimum makespan scheduling problem is to find an assignment of tasks to servers that minimizes the completion time of the task that finishes last. This time is called the makespan of the assignment. We consider the online version of the problem under the restricted assignment model, where tasks present sequentially and each of them is associated with a subset of feasible servers. Each task needs to be assigned to a feasible server upon its arrival and the decision cannot be revoked. Formally,

Definition 3.2 (Online Restricted Assignment System) *Let S be a set of* servers*, T be a set of* tasks*, and $w : T \to \mathbb{R}^+$ be a* cost function*. Let π be a* permutation *of tasks in T to represent the order of arrival. Let $\rho \subseteq T \times S$ be a* data placement relation *that determines the subset of* feasible servers *for each task t. An* online restricted assignment system (ORA-system) *is a five-tuple $\sigma = (S, T, w, \pi, \rho)$. The* total load *of σ is defined as $H(\sigma) = \sum_{t \in T} w(t)$.*

Similar to the discussions in Chapter 2, although the cost function in an ORA-system includes real numbers, in order to avoid the issues of number representation and precision, we consider rational costs instead in the analysis of algorithms. To make one step further, because rational costs can be easily normalized by multiplying a proper integer, it is straightforward to convert rational costs to integers. The normalization does not affect our competitive analysis, and the main theorem on competitive ratio bound remains the same. Therefore, we focus on integer task costs in the remaining of this chapter.

An ORA-system can be represented by a weighted bipartite graph, with one side of vertices representing tasks and the other side representing servers. Figure 3.1 shows an example of a scheduling problem with 4 tasks and 2 servers. Each task is connected to its feasible servers and the weight besides each task represents its cost. The subscript of a task represents its order of arrival.

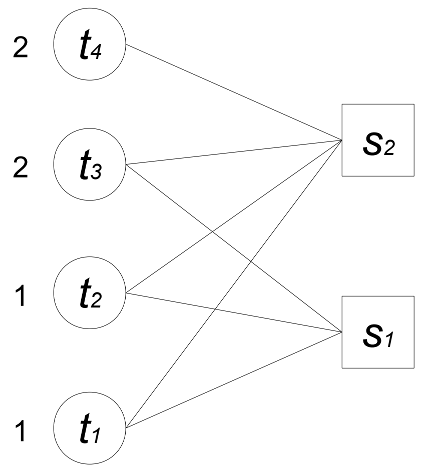

Figure 3.1: An ORA-system with 4 tasks and 2 servers.

An online algorithm is given one task at a time. It assigns the current task to a feasible server without the knowledge about the forthcoming tasks. The goal is to minimize the resulting makespan when all tasks are assigned. The problem is defined formally below.

Problem 3.2 Online Minimum Makespan Scheduling with Restricted Assignment

1. **Instance:** An ORA-system $\sigma = (S, T, w, \pi, \rho)$.
2. **Objective:** Compute a restricted task assignment A online that minimizes L^A.

Because we consider only restricted task assignment in this chapter, we will simplify the notation and use assignment to mean restricted assignment in the rest of our discussion.

An assignment can be naturally represented by a bipartite graph. An edge (t, s) indicates that task t is assigned to server s. Figure 3.2 shows two possible assignments in the ORA-system shown in Figure 3.1.

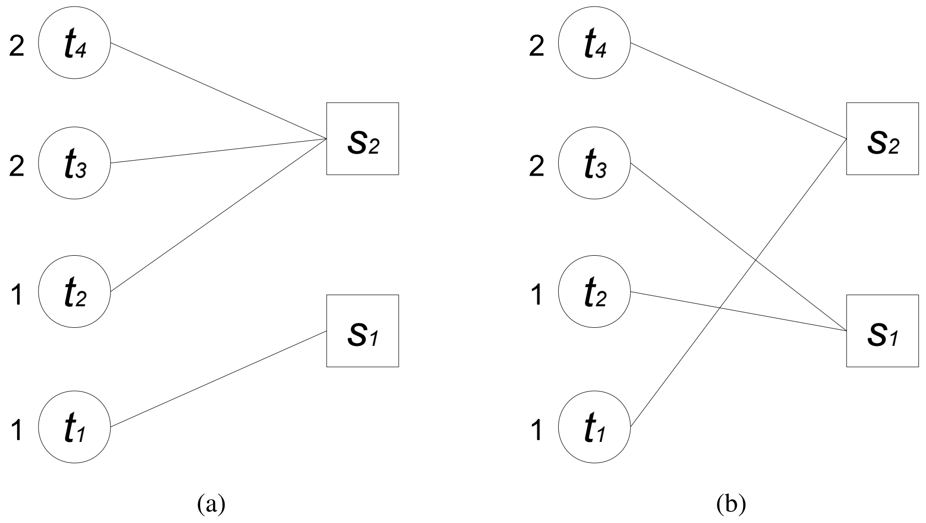

Figure 3.2: (a) Assignment A, (b) Assignment B.

3.4 Competitive Analysis of GREEDY

In this section, we define GREEDY, the algorithm under study. The main theorems on total load and competitive ratio bounds are also presented.

3.4.1 The GREEDY Algorithm

A natural algorithm, GREEDY, works by assigning each task to any feasible server with the minimum load. Ties are broken arbitrarily. The performance of GREEDY is compared against the best possible offline algorithm OPT that knows the sequence in advance and has unbounded computational ability. Assignments A and B in Figure 3.2 are possible assignments computed by GREEDY and OPT, respectively, in the ORA-system shown in Figure 3.1. The ratio between makespan of the two assignments is $5/3$. Given an

ORA-system σ, let A_σ be an assignment computed by GREEDY and O_σ be an assignment computed by OPT. The competitive ratio of GREEDY is defined as the maximum of $L^{A_\sigma}/L^{O_\sigma}$ over all possible σ.

Traditionally, an upper bound on the competitive ratio is obtained by fixing the parameters of the ORA-system and analyzing the competitive ratio directly in this system. We develop an alternative approach for competitive analysis. We first fix the makespans of assignments computed by GREEDY and OPT, and bound from below the total task load of any ORA-systems leading to such makespans. Then we link the total load with competitive ratio by the simple fact that any makespan is no better than the average server load, i.e., the total load divided by the number of servers. This step translates the lower bound on total task load into an upper bound on the competitive ratio.

We begin the analysis of total task load by partitioning the problem space with makespan values. Given two parameters $d, \ell \in \mathbb{Z}^+$, let $\mathcal{N}(d, \ell)$ be the set of ORA-systems such that GREEDY can produce assignment with makespan d and OPT produces assignment with makespan no greater than ℓ. The *minimum total load problem* is to determine the minimum total load of any ORA-system in $\mathcal{N}(d, \ell)$.

Problem 3.3 The Minimum Total Load Problem

1. **Instance:** $d, \ell \in \mathbb{Z}^+$.
2. **Objective:** Determine $\min\{H(\sigma) : \sigma \in \mathcal{N}(d, \ell)\}$.

3.4.2 Main Results

Our main result on the upper-bound for competitive ratio of GREEDY is presented in Theorem 3.2. The proof is based on the lower bound result on total load shown in Theorem 3.3. The rest of this chapter is devoted to the proof of Theorem 3.3.

Theorem 3.2 (Upper bound on competitive ratio of GREEDY) *For any ORA-system σ, let n be the number of servers, A_σ and O_σ be assignments computed by GREEDY and OPT, respectively. Then*

$$\frac{L^{A_\sigma}}{L^{O_\sigma}} \leq \log n + 1 - \delta(\sigma),$$

where $0 \leq \delta(\sigma) \leq \log n$. $\delta(\sigma) = 0$ if and only if O_σ perfectly balances load over all servers and L^{A_σ} is a multiple of L^{O_σ}.

Theorem 3.3 (Lower bound on total load) *For any ORA-system, let H be the total task load, d be the makespan of an assignment computed by GREEDY, and ℓ be the makespan of an optimal assignment. Then*

$$H \geq 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$$

We remark that when $\ell = 1$, Theorem 3.3 reduces to $H \geq 2^{d-1}$. This result is relevant to the analysis of the tree structure for solving equivalence problems [33]. Assume a merge is done by attaching the tree with the smaller number of vertices to the root of the tree with larger number. With some minor change, our proof of Theorem 3.3 also shows that, in such a tree of height $d - 1$, the number of total vertices is at least 2^{d-1}. Therefore, for a set of n items, each find operation on this tree structure can be performed in $O(\log n)$ time.

To prove Theorem 3.2, we translate the lower bound on total load in Theorem 3.3 into an upper bound for the competitive ratio of GREEDY.

Lemma 3.4 *For any ORA-system, let H be the total task load, d be the makespan of an assignment computed by GREEDY, and ℓ be the makespan of an optimal assignment. Then*

$$\frac{d}{\ell} \leq \log\left(\frac{H}{\ell}\right) + \left(1 + \frac{d \bmod \ell}{\ell}\right) - \log\left(1 + \frac{d \bmod \ell}{\ell}\right)$$

Proof. By Theorem 3.3,

$$H \geq 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$$

Dividing both sides by ℓ gives

$$\frac{H}{\ell} \geq 2^{(\lfloor d/\ell \rfloor - 1)} \left(1 + \frac{d \bmod \ell}{\ell}\right)$$

Taking logarithm on both sides and rearranging terms gives

$$\left\lfloor \frac{d}{\ell} \right\rfloor \leq \log\left(\frac{H}{\ell}\right) + 1 - \log\left(1 + \frac{d \bmod \ell}{\ell}\right)$$

Therefore,

$$\begin{aligned} \frac{d}{\ell} &= \left\lfloor \frac{d}{\ell} \right\rfloor + \frac{d \bmod \ell}{\ell} \\ &\leq \log\left(\frac{H}{\ell}\right) + \left(1 + \frac{d \bmod \ell}{\ell}\right) - \log\left(1 + \frac{d \bmod \ell}{\ell}\right) \end{aligned}$$

$\square$

Proof of Theorem 3.2: Let $H = H(\sigma)$, $d = L^{A_\sigma}$, and $\ell = L^{O_\sigma}$. Let

$$f(x) = (1 + x) - \log(1 + x) \tag{3.1}$$

$$\delta(\sigma) = \log n - \log\left(\frac{H}{\ell}\right) + 1 - f\left(\frac{d \bmod \ell}{\ell}\right) \tag{3.2}$$

By Lemma 3.4,

$$\frac{L^{A_\sigma}}{L^{O_\sigma}} \leq \log\left(\frac{H}{\ell}\right) + \left(1 + \frac{d \bmod \ell}{\ell}\right) - \log\left(1 + \frac{d \bmod \ell}{\ell}\right) \tag{3.3}$$

Combining (3.1), (3.2), and (3.3) gives

$$\frac{L^{A_\sigma}}{L^{O_\sigma}} \le \log n + 1 - \delta(\sigma)$$

Now we bound the function $\delta(\sigma)$. For $x \in [0, 1)$, $f(x) = 1$ if $x = 0$, and $0 < f(x) < 1$ otherwise. Then

$$0 < f\left(\frac{d \bmod \ell}{\ell}\right) \le 1, \tag{3.4}$$

where the equality holds if and only if d is a multiple of ℓ. Note that the optimal makespan is no smaller than the average server load, i.e.,

$$\ell \ge H/n,$$

and the equality holds if and only if O_σ perfectly balances load over all servers. Therefore,

$$\log n \ge \log\left(\frac{H}{\ell}\right) \tag{3.5}$$

Combining (3.2), (3.4), and (3.5) gives the lower bound for $\delta(\sigma)$,

$$\delta(\sigma) \ge 0, \tag{3.6}$$

where the equality holds if and only if O_σ perfectly balances load over all servers and d is a multiple of ℓ. To upper-bound $\delta(\sigma)$, we follow a case by case analysis. Note that $H/\ell \ge 1$ because the total load H is no smaller than any makespan. When $H/\ell = 1$, it follows that $\ell = d$ because $H \ge d$ and $d \ge \ell$. Thus

$$\delta(\sigma) = \log n - \log 1 + 1 - f(0) = \log n$$

When $H/\ell \geq 2$, we have

$$\delta(\sigma) \leq \log n - \log 2 + 1 - f\left(\frac{d \bmod \ell}{\ell}\right) < \log n$$

When $1 < H/\ell < 2$, if $d = \ell$, then

$$\delta(\sigma) < \log n - \log 1 + 1 - f(0) < \log n$$

If $d > \ell$, it follows that $\ell < d < 2\ell$ because $d \leq H < 2\ell$. Then

$$\begin{aligned}
& \log\left(\frac{H}{\ell}\right) + f\left(\frac{d \bmod \ell}{\ell}\right) \\
\geq\ & \log\left(\frac{d}{\ell}\right) + f\left(\frac{d \bmod \ell}{\ell}\right) \\
=\ & \log\left(1 + \frac{d \bmod \ell}{\ell}\right) + \left(1 + \frac{d \bmod \ell}{\ell}\right) - \log\left(1 + \frac{d \bmod \ell}{\ell}\right) \\
>\ & 1
\end{aligned}$$

Therefore, When $1 < H/\ell < 2$ and $d > \ell$, we have

$$\delta(\sigma) < \log n + 1 - 1 = \log n$$

In summary, we have shown in all possible cases that

$$\delta(\sigma) \leq \log n \tag{3.7}$$

Combining (3.6) and (3.7), we conclude that

$$0 \leq \delta(\sigma) \leq \log n$$

□

We remark that this bound refines the original result of Azar, Naor, and Rom [6]. Compared to the best result known to date by Buchbinder and Naor [12], this bound is better in some ORA-systems, but is not as good in some other ORA-systems.

3.5 Witness Graphs

In this section, we define and construct *witness graphs* to embed a pair of assignments in an ORA-system, allowing them to be compared in the same framework. A witness graph is an ordered weighted multidigraph. Parallel edges that share the same pair of vertices are distinguished and allowed to carry different weights.

3.5.1 Definition of Witness Graph

Definition 3.3 (Witness Graph) *Let V be a set of* vertices*, E be a multiset of* directed edges *(or simply* edges*) between vertices in V, $w : E \to \mathbb{R}^+$ be a* weight function*, and ϕ be a* permutation *of edges in E. A* witness graph *is an ordered weighted multidigraph $G = (V, E, w, \phi)$. The* weight *of G is defined as $W(G) = \sum_{e \in E} w(e)$.*

Definition 3.4 (Indegree and outdegree) *Given a witness graph $G = (V, E, w, \phi)$, for each vertex $v \in V$, define its* indegree *$d_{\mathrm{i}}(v)$ as the total weight of incoming edges and its* outdegree *$d_{\mathrm{o}}(v)$ as the total weight of outgoing edges. In other words,*

$$d_{\mathrm{i}}(v) = \sum_{e=(u,v)} w(e), \quad d_{\mathrm{o}}(v) = \sum_{e=(v,u)} w(e)$$

The maximum indegree *and* maximum outdegree *of G are defined as $d_{\mathrm{i}}^G = \max_v d_{\mathrm{i}}(v)$ and $d_{\mathrm{o}}^G = \max_v d_{\mathrm{o}}(v)$, respectively.*

3.5.2 Embedding Assignments in Witness Graphs

We describe how to construct a witness graph to embed two assignments in an ORA-system. We first introduce a general approach to embed two functions $f : X \to Y$ and $g : X \to Y$ over finite sets X and Y into a multidigraph. Then we extend the multidigraph to includes weight and ordering information, and thus encode task costs and ordering in the resulting witness graph.

Embedding Functions

Given two functions $f : X \to Y$ and $g : X \to Y$ where X, Y are finite sets, we construct a multidigraph (V, E) by creating one-to-one correspondence between Y and V, X and E, respectively. Each element in Y corresponds to a distinct vertex in V, and each element $x \in X$ corresponds to a distinct directed edge (u, v), where u corresponds to $g(x)$ and v corresponds to $f(x)$. Formally,

Definition 3.5 (Multidigraph from functions) *Given two functions $f : X \to Y$ and $g : X \to Y$ where X, Y are finite sets, construct a multidigraph graph $G = (V, E)$ with the vertex set $V = Y$ and the edge multiset $E = \{(g(x), f(x)) \mid x \in X\}$.*

Functions can be represented by bipartite graphs. We visualize the construction of a multidigraph from the two assignments A and B shown in Figure 3.2. Note both assignments are functions that map T to S. As shown in Figure 3.3, in the first step, we add directions on the undirected edges in the two bipartite graphs, such that in assignment A edges go from tasks to servers and in assignment B edges go from servers to tasks. Then, we merge the two copies of each task vertex into one and keep the edges. Next, we eliminate task vertices entirely. The two directed edges used to connect a task vertex to server vertices now become a single directed edge. Finally, we merge the two copies of each server vertex. This last step gives a multidigraph.

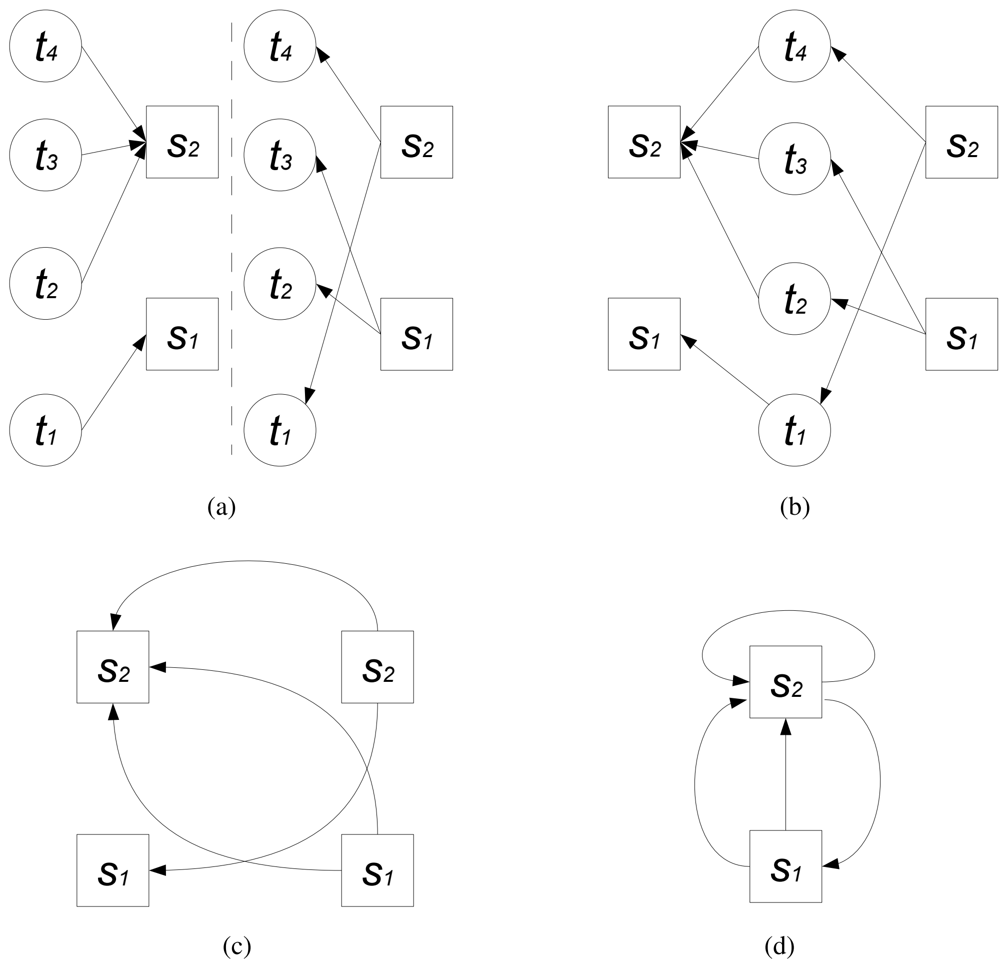

Figure 3.3: (a) Step 1: add directions on edges, (b) Step 2: merge task vertices, (c) Step 3: eliminate task vertices, (d) Step 4: merge server vertices.

It is not hard to see how the other direction from a multidigraph to two functions works. The construction is as follows.

Definition 3.6 (Functions from multidigraph) *Given a multidigraph $G = (V, E)$, construct two sets $X = E$ and $Y = V$. Define two functions $f : X \to Y$ and $g : X \to Y$ such that, for any element $x \in X$, let $f(x) = v$ and $g(x) = u$ where $x = (u, v)$.*

We have the following fact on the mappings defined in Definitions 3.5 and 3.6.

Fact 3.5 *The mapping from two functions f, g to a multidigraph G in Definition 3.5 is the inverse of the mapping from a multidigraph G to two functions f, g in Definition 3.6.*

Therefore, a multidigraph is a re-interpretation of two functions. The structure of both functions are preserved in the corresponding multidigraph.

Embedding Assignments

Now we define the mapping from two assignments in an ORA-system to a witness graph. Because assignments are functions that map T to S, by Definition 3.5, a multidigraph (V, E) can be constructed from a pair of assignments such that each vertex $v \in V$ corresponds to a distinct server $s \in S$, and each edge $e \in E$ corresponds to a distinct task $t \in T$. Therefore, it is natural to augment the multidigraph with edge weights and edge ordering to reflect the cost and order of arrival of tasks. The augmented graph is a witness graph. For example, Figure 3.4 is a witness graph by augmenting the multidigraph in Figure 3.3(d) with edge weights (the value besides each edge) and edge ordering (the circled value besides each edge).

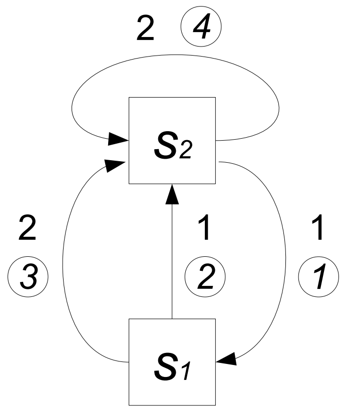

Figure 3.4: A witness graph.

Formally, the construction of a witness graph from two assignments in an ORA-system is defined as follows.

Definition 3.7 (Witness graph from assignments) *Given two task assignments A and B in an ORA-system $\sigma = (S, T, w, \pi, \rho)$, construct a witness graph $G_\sigma^{A,B} = (V, E, w', \phi)$*

with the vertex set $V = S$, the edge multiset $E = \{(B(t), A(t)) \mid t \in T\}$, the edge weight $\{w'((B(t), A(t))) = w(t) \mid t \in T\}$, and the permutation ϕ of edges in E that is compatible with π. To simplify the notation, we may omit σ, A and B from $G_\sigma^{A,B}$ if they are obvious in the context.

The following lemma is immediate from Definition 3.7.

Lemma 3.6 *Let A, B be two assignments in an ORA-system $\sigma = (S, T, w, \pi, \rho)$, and $G = (V, E, w', \phi)$ be a witness graph constructed from (σ, A, B) following Definition 3.7. Then S and T are in one-to-one correspondence with T and E, respectively. For task t that is corresponding to edge (u, v), it holds that $A(t) = v$, $B(t) = u$, $w(t) = w'((u, v))$, and $\pi(t) = \phi((u, v))$. For server s that is corresponding to vertex v, it holds that $L_s^A = \mathrm{d_i}(v)$ and $L_s^B = \mathrm{d_o}(v)$. Therefore, $H(\sigma) = W(G)$, $L^A = \mathrm{d}_\mathrm{i}^G$, and $L^B = \mathrm{d}_\mathrm{o}^G$.*

If t is a task that corresponds to an edge (u, v), when further confusions cannot be made, we slightly abuse the notation and write $t = (u, v)$. Similarly, if s is a server that corresponds to a vertex v, we write $s = v$.

3.5.3 Minimum Graph Weight Problem

For an edge e in a witness graph, let u be its starting vertex and v be its ending vertex. Let E_e be the set of edges that point to v and appear earlier than e in the permutation ϕ. Define the *pre-weight* of edge e to be $\eta(e) = \sum_{e' \in E_e} w(e')$. Then the witness graph $G_\sigma^{A,B}$ where A is computed by GREEDY has the following property.

Fact 3.7 *Let $G_\sigma^{A,B}$ be a witness graph constructed from two assignments A and B in an ORA-system where A is computed by GREEDY. Then $\mathrm{d_i}(u) \geq \eta(e)$ for all edges $e = (u, v)$ in $G_\sigma^{A,B}$.*

Proof. To simplify the notation, we write $G = G_\sigma^{A,B}$. Let v be a vertex with incoming edges $e_0, \cdots, e_{k-1}$. Then each edge $e_j = (v_j, v)$ corresponds to a distinct task and the k edges are ordered according to the order of task arrivals. By the definition of GREEDY, for any j between 0 and $k-1$, at the moment when task e_j is to be assigned to server v, the load of server v_j must be no smaller than that of server v, which is the total cost of tasks assigned to v before task e_j and is equal to $\eta(e_j)$. Otherwise GREEDY would have assigned task e_j to server v_j instead. By Lemma 3.6, the load of server v_j under assignment A is equal to the indegree of vertex v_j, i.e., $L_{v_j}^A = d_{\mathrm{i}}(v_j)$. Therefore, $d_{\mathrm{i}}(v_j) \geq \eta(e_j)$. This fact holds for any $v \in V$ and any $e \in E$, and thus the claim is proved. $\square$

To lower-bound the minimum total load of ORA-systems in $\mathcal{N}(d, \ell)$, we define a family of witness graphs $\mathcal{G}(d, \ell)$ and show that a lower bound on the weight of witness graphs in $\mathcal{G}(d, \ell)$ is a lower bound on the total load of ORA-systems in $\mathcal{N}(d, \ell)$.

Given two parameters $d, \ell \in \mathbb{Z}^+$, let $\mathcal{G}(d, \ell)$ be the set of witness graphs such that for any $G \in \mathcal{G}(d, \ell)$, it holds that $d_{\mathrm{i}}^G = d$, $d_{\mathrm{o}}^G \leq \ell$, and $d_{\mathrm{i}}(u) \geq \eta(e)$ for all edges $e = (u, v)$. There are potentially many different witness graphs in $\mathcal{G}(d, \ell)$. We are interested in those have the smallest weight. The *minimum graph weight problem* is to determine the minimum weight of any graph in $\mathcal{G}(d, \ell)$.

Problem 3.4 The Minimum Graph Weight Problem

1. **Instance:** $d, \ell \in \mathbb{Z}^+$.
2. **Objective:** Determine $\min\{W(G) : G \in \mathcal{G}(d, \ell)\}$.

The first observation is the interesting connection between the S_n trees for $n = d$ defined in [27] and the class of witness graphs in $\mathcal{G}(d, 1)$. S_n trees are defined recursively: any tree consisting of a single vertex is an S_0 tree, and the tree obtained by attaching the root of an S_{n-1} tree to the root of another disjoint S_{n-1} tree is an S_n tree. Now we add extras to an S_d tree. (1) Edge direction: all edges point towards the root. (2) Edge ordering: let e^i and e^j be any two edges that leave vertices of height i and j, respectively. Then let the ordering $\phi(e^i) < \phi(e^j)$ iff $i < j$. When $i = j$, the ordering between the two is arbitrary. (3) Self-loop: the root of the tree is a unique vertex that has no outgoing edge. Add a new edge as the self-loop of the root and make it appear last in the edge ordering. (4) Edge weight: all edges have weight 1. We call the resulting graph S'_d. The observation is that S'_d is a witness graph of minimum weight in $\mathcal{G}(d, 1)$. Thus, our analysis on $\mathcal{G}(d, \ell)$ can be applied to the analysis of S_d trees.

The second observation is that the minimum graph weight problem under the condition that $d \le \ell$ is trivial, because a graph having a single vertex with a self-loop of weight d is a witness graph in $\mathcal{G}(d, \ell)$. Such a graph has the smallest weight among all witness graphs in $\mathcal{G}(d, \ell)$ because any graph $G \in \mathcal{G}(d, \ell)$ has at least one vertex of indegree d, leading to the result that $W(G) \ge d$ when $d \le \ell$. The more interesting case is when $d > \ell$.

We establish the following lemma that a lower bound for the minimum graph weight is a lower bound for the minimum total load problem.

Lemma 3.8 *For any* $d, \ell \in \mathbb{Z}^+$, $\min_{G \in \mathcal{G}(d,\ell)} W(G) \le \min_{\sigma \in \mathcal{N}(d,\ell)} H(\sigma)$.

Proof. Let $\sigma \in \mathcal{N}(d, \ell)$, A_σ be an output of GREEDY and O_σ be an output of OPT such that $L^{A_\sigma} = d, L^{O_\sigma} \le \ell$. Let $G = G_\sigma^{A_\sigma, O_\sigma}$ be the witness graph constructed from $(\sigma, A_\sigma, B_\sigma)$. By Lemma 3.6, $W(G) = H(\sigma)$, $d_{\mathrm{i}}^G = L^{A_\sigma}$ and $d_{\mathrm{o}}^G = L^{O_\sigma}$. Therefore, $d_{\mathrm{i}}^G = d$ and $d_{\mathrm{o}}^G \le \ell$, and thus $G \in \mathcal{G}(d, \ell)$. Let $\mathcal{G}' = \{G_\sigma^{A_\sigma, O_\sigma} : \sigma \in \mathcal{N}(d, \ell)\}$. Then $\mathcal{G}' \subseteq \mathcal{G}(d, \ell)$. It follows that

$$\min_{G \in \mathcal{G}(d,\ell)} W(G) \le \min_{G \in \mathcal{G}'} W(G) = \min_{\sigma \in \mathcal{N}(d,\ell)} H(\sigma)$$

□

3.6 Token Production Systems (TPSs)

3.6.1 Definition of TPS

In this section, we first define token production system (TPS). Then we define the *membership problem* and the *vector weight problem* based on the algebraic structure of TPSs, and show that they are equivalent to the *reachability problem* and the *min-cost reachability problem* with *activities*. We also define the *1-stop reachability problem* and discuss the linear programming formulation of the optimization problems.

A token production system is a finite set of vectors, each associated with a cost.

Definition 3.8 (Token Production System) *A* token production system (TPS) *of dimension $d \in \mathbb{Z}^+$ is a tuple (U, C), where $U \subset \mathbb{Q}^d$, $|U| \in \mathbb{N}$, and $C : U \to \mathbb{Q}$. U is the* basis set *and C is the* cost function.

In the following discussion, if the dimension of a TPS is not explicitly stated, we assume that its dimension is $d \in \mathbb{Z}^+$. Given a TPS (U, C), the conical hull of U is

$$coni(U) = \left\{ \sum_i r_i \boldsymbol{u}_i : r_i \geq 0, \boldsymbol{u}_i \in U \right\}$$

By convention, define $\min \emptyset = \infty$, where $\emptyset$ is the empty set. Then we define the *weight* for any $\boldsymbol{p} \in \mathbb{Q}^d$ as

$$\omega(\boldsymbol{p}) = \min \left\{ \sum_i r_i C(\boldsymbol{u}_i) : r_i \geq 0, \boldsymbol{u}_i \in U, \sum_i r_i \boldsymbol{u}_i = \boldsymbol{p} \right\}$$

Below are two simple facts about the weight of vectors with scalar multiplication and vector addition.

Fact 3.9 *For any TPS, any vector $\mathbf{p}$, and any $r \geq 0$, it holds that $\omega(r\boldsymbol{p}) = r \cdot \omega(\boldsymbol{p})$.*

Fact 3.10 *For any TPS, any vector $\boldsymbol{p}$ and $\boldsymbol{q}$, it holds that $\omega(\boldsymbol{p}+\boldsymbol{q}) \le \omega(\boldsymbol{p}) + \omega(\boldsymbol{q})$.*

We define the *membership problem* and *vector weight problem* below.

Problem 3.5 The membership problem

1. **Instance:** A TPS (U, C) of dimension $d \in \mathbb{Z}^+$ and a vector $\boldsymbol{p} \in \mathbb{Q}^d$.
2. **Objective:** Decide whether $\boldsymbol{p} \in coni(U)$.

Problem 3.6 The vector weight problem

1. **Instance:** A TPS (U, C) of dimension $d \in \mathbb{Z}^+$ and a vector $\boldsymbol{p} \in \mathbb{Q}^d$.
2. **Objective:** Determine $\omega(\boldsymbol{p})$.

An *action* is a tuple $a = (\boldsymbol{u}, r)$ where $\boldsymbol{u} \in U$ and $r \in \mathbb{Q}^+$. We call $\boldsymbol{u}(a)$ the vector of action a and $r(a)$ the scalar of a. We extend the cost definition for an action a to be $C(a) = rC(\boldsymbol{u})$. An *activity* M is a finite sequence of actions $(a_1, \cdots, a_k)$. We say the vector of M is $\boldsymbol{v}(M) = \sum_i r_i \boldsymbol{u}_i$ and extend its cost to be $C(M) = \sum_i r_i C(\boldsymbol{u}_i)$. Note that $C(M) \ge \omega(\boldsymbol{v}(M))$ by definition. M is *optimal* if $C(M) = \omega(\boldsymbol{v}(M))$. Applying an activity M to a vector $\boldsymbol{p}$ gives a new vector as the result of vector addition $\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{v}(M)$. We write $\boldsymbol{p} \xrightarrow{M} \boldsymbol{p}'$.

With the definition of activity, we define the *reachability problem* and *min-cost reachability problem* below, which are equivalent to the membership problem and the vector weight problem for TPSs, respectively.

Problem 3.7 The reachability problem

1. **Instance:** A TPS (U, C) of dimension $d \in \mathbb{Z}^+$ and a vector $\boldsymbol{p} \in \mathbb{Q}^d$.
2. **Objective:** Decide whether there exists an activity M such that $\mathbf{0} \xrightarrow{M} \boldsymbol{p}$.

We state several simple facts without proof below.

Problem 3.8 The min-cost reachability problem

1. **Instance:** A TPS (U, C) of dimension $d \in \mathbb{Z}^+$ and a vector $\boldsymbol{p} \in \mathbb{Q}^d$.
2. **Objective:** Determine $\min\{C(M) : \mathbf{0} \xrightarrow{M} \boldsymbol{p}\}$.

Fact 3.11 *For any TPS (U, C) and vector $\mathbf{p}$, it holds that $\mathbf{p} \in coni(U)$ if and only if there exists an activity M such that $\mathbf{0} \xrightarrow{M} \boldsymbol{p}$.*

Fact 3.12 *For any TPS and vector $\mathbf{p}$, it holds that $\omega(\boldsymbol{p}) = \min\{C(M) : \mathbf{0} \xrightarrow{M} \boldsymbol{p}\}$.*

Fact 3.13 *For any TPS, vectors $\mathbf{p}, \mathbf{p}'$, and activity M, it holds that $\boldsymbol{p} \xrightarrow{M} \boldsymbol{p}'$ if and only if $\mathbf{0} \xrightarrow{M} \boldsymbol{p}' - \boldsymbol{p}$ and $\min\{C(M) : \boldsymbol{p} \xrightarrow{M} \boldsymbol{p}'\} = \min\{C(M) : \mathbf{0} \xrightarrow{M} \boldsymbol{p}' - \boldsymbol{p}\}$.*

There is a straightforward linear programming formulation of the min-cost reachability problem, which is stated below.

Lemma 3.14 *The min-cost reachability problem for a TPS can be formulated as a linear programming problem and be solved in time polynomial in the dimension of the problem and the number of bits in the input.*

Proof. Let $\boldsymbol{x} = (r_1, \cdots, r_k)^T$, $\boldsymbol{c} = (C(\boldsymbol{u}_1), \cdots, C(\boldsymbol{u}_k))$, $A = (\boldsymbol{u}_1^T, \cdots, \boldsymbol{u}_k^T)$, where $\{\boldsymbol{u}_1, \cdots, \boldsymbol{u}_k\} = U$. Then the min-cost reachability problem can be rewritten as the following LP:

$$\begin{aligned} \min \quad & \boldsymbol{c}\boldsymbol{x} \\ s.t. \quad & A\boldsymbol{x} = \boldsymbol{p}, \\ & \boldsymbol{x} \geq \mathbf{0}. \end{aligned}$$

The minimum reachability cost equals the optimal value of this LP. As LPs can be solved by interior point method [46] in time polynomial in the dimension of the problem and the number of bits in the input, the min-cost reachability problem for TPSs can also be solved in polynomial time. □

In some applications, it is important to first reach some critical intermediate states before going to the final state. To model such problems, we define the *1-stop reachability problem* and its min-cost version below.

Problem 3.9 The 1-stop reachability problem

---

1. **Instance:** A TPS (U, C) of dimension $d \in \mathbb{Z}^+$, a finite set of vectors $V \subset \mathbb{Q}^d$, and a vector $\boldsymbol{p} \in \mathbb{Q}^d$.
2. **Objective:** Decide whether there exist activities M_1, M_2 such that $\mathbf{0} \xrightarrow{M_1} \boldsymbol{q} \xrightarrow{M_2} \boldsymbol{p}$ for some vector $\boldsymbol{q} \in V$.

Let $M_1 = (a_1, \cdots, a_i)$ and $M_2 = (b_1, \cdots, b_j)$ be two activities. Define their *concatenation* to be $M_1 \| M_2 = (a_1, \cdots, a_i, b_1, \cdots, b_j)$. In other words, the concatenation of two activities is another activity.

Problem 3.10 The min-cost 1-stop reachability problem

---

1. **Instance:** A TPS (U, C) of dimension $d \in \mathbb{Z}^+$, a finite set of vectors $V \subset \mathbb{Q}^d$, and a vector $\boldsymbol{p} \in \mathbb{Q}^d$.
2. **Objective:** Determine $\min\{C(M_1 \| M_2) : \mathbf{0} \xrightarrow{M_1} \boldsymbol{q} \xrightarrow{M_2} \boldsymbol{p}, \boldsymbol{q} \in V\}$.

Lemma 3.15 *The min-cost 1-stop reachability problem for a TPS can be formulated as $|V|$ linear programming problems and be solved in time polynomial in the dimension of the problem and the number of bits in the input.*

Proof. For each $\boldsymbol{q} \in V$, we formulate a linear programming problem and compute an optimal value. The minimum over all the $|V|$ optimal values gives answer to the min-cost 1-stop reachability problem. □

Apparently, min-cost with the 1-stop requirement is at least as large as the one without such a requirement, because 1-stop activities are forced to detour to go through V. The two costs are equal if and only if some min-cost activity passes through V. In this sense, the min-cost reachability in TPSs satisfies the triangle inequality.

3.6.2 Properties of TPSs

In this section, we discuss properties of TPSs based on restrictions on the vector sets. At the top level, we have defined the general TPSs in Section 3.6.1. Going one level down, we discuss *directional* TPSs based on the notion of *directional* vector sets. We further restrict the set of directional vectors and define the *covering* vector sets, which give rise to the *covering* TPSs. Finally, we describe a way of constructing *bounded* vector sets and define the *bounded* TPSs accordingly. The following statement on reachability over superset of vectors is straightforward.

Fact 3.16 *For any vector sets $U \subseteq V$ and any vector $\mathbf{p}$, if $\mathbf{p}$ is reachable by activities generated from vectors in U, then $\mathbf{p}$ is reachable by activities generated from vectors in V.*

Definition 3.9 (Directional vector sets and TPSs) *A set of vectors U are* directional *if any vector $\boldsymbol{u} \in U$ is either in the form $(\boldsymbol{u}[0], \boldsymbol{u}[1], \cdots, \boldsymbol{u}[i-1], \boldsymbol{u}[i], 0, \cdots, 0)$ where $\boldsymbol{u}[0], \cdots, \boldsymbol{u}[i-1] \leq 0$ and $\boldsymbol{u}[i] > 0$, or in the form of its negation. Vectors of the first form are called* productions for bucket i *and the second ones are called* consumptions for bucket i. *A directional TPS is a TPS (U, C) where U is directional.*

Given a directional TPS (U, C) of dimension d, for all $0 \leq i \leq d-1$, let $U(i)$ be the subset of all productions and consumptions for bucket i. We define the covering vectors as a further restricted version of the directional vector sets, and then define the covering TPSs accordingly.

Definition 3.10 (Covering vector sets and TPSs) *A set of directional vectors U are* covering *if $U(i)$ contains both production and consumption for all $0 \leq i \leq d-1$. A covering TPS is a TPS (U, C) where U is covering.*

Now we describe the most restricted class of vector sets we study in this chapter. These vector sets are subsets of $\mathbb{Z}^d$ and the costs associated with these vectors are in $\mathbb{Z}$. The definition of these vector sets is based on integer partitions.

For any positive integer $x \in \mathbb{Z}^+$, define an *integer partition* of x to be a sequence of positive integers that sum to x. An *ℓ-bounded integer partition I_x* of x is an integer partition such that each integer in the sequence is no greater than ℓ. The parameter ℓ is a positive integer and is called the *width* of the integer partition. Let $\mathcal{I}_x$ be the set of all possible ℓ-bounded integer partitions for x. Among all members in the set $\mathcal{I}_x$, there is a particular one called the *ℓ-regular integer partition* for x,

$$I_x^* = \begin{cases} (\ell, \cdots, \ell) & \text{if } \ell \mid x, \\ (x \bmod \ell, \ell, \cdots, \ell) & \text{otherwise.} \end{cases}$$

When x is obvious in the context, we omit the subscript and write I and I^*. By convention, define (0) to be the ℓ-bounded integer partition for 0.

Based on ℓ-bounded integer partitions, we define three sets of vectors in $\mathbb{Z}^d$ below.

- Production ($\boldsymbol{\alpha}$ vectors):

$$\boldsymbol{\alpha}_{i,I} = (-i_0, \underbrace{0, \cdots, 0}_{i_0-1}, \cdots, -i_k, \underbrace{0, \cdots, 0}_{i_k-1}, +\ell, \underbrace{0, \cdots, 0}_{d-1-i}),$$

  where $0 \leq i \leq d-1$ and $I = (i_0, \cdots, i_k) \in \mathcal{I}_i$. We call $\boldsymbol{\alpha}_{i,I^*}$ the *regular* production for i. To simplify the notation, we use $\boldsymbol{\alpha}_i$ to represent a vector $\boldsymbol{\alpha}_{i,I}$ with an arbitrary $I \in \mathcal{I}_i$ and $\boldsymbol{\alpha}_i^*$ to represent $\boldsymbol{\alpha}_{i,I^*}$.

- Transfer ($\boldsymbol{\beta}$ vectors):

$$\boldsymbol{\beta}_i = (\underbrace{0, \cdots, 0}_{i-1}, +1, -1, \underbrace{0, \cdots, 0}_{d-1-i}),$$

  where $1 \leq i \leq d-1$.

- Redemption ($\boldsymbol{\gamma}$ vectors):

$$\boldsymbol{\gamma}_i = (\underbrace{0, \cdots, 0}_{i}, -1, \underbrace{0, \cdots, 0}_{d-1-i}),$$

  where $0 \leq i \leq d-1$.

Intuitively, a production creates ℓ units of supplies in bucket i and i units of demands in lower indexed buckets in a way as described by the integer partition I. A transfer creates 1 unit of demand in bucket i and 1 unit of supply in bucket $i-1$. A redemption creates 1 unit of demand in bucket i.

Define the *ℓ-bounded* vector set as $U_\ell = \{\boldsymbol{\alpha}_{i,I}, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i : 0 \leq i \leq d-1, I \in \mathcal{I}_i\}$ and the *ℓ-regular* vector set as $U^*_\ell = \{\boldsymbol{\alpha}^*_i, \boldsymbol{\beta}_i, \boldsymbol{\gamma}_i : 0 \leq i \leq d-1\}$. Define the *$\ell$-bounded* cost function C_ℓ as

$$C_\ell(\boldsymbol{u}) = \begin{cases} \ell & \text{if } \boldsymbol{u} \text{ is production,} \\ 0 & \text{if } \boldsymbol{u} \text{ is transfer,} \\ -1 & \text{if } \boldsymbol{u} \text{ is redempton.} \end{cases}$$

Definition 3.11 (Bounded and regular TPSs) *A bounded TPS of width ℓ is a TPS $\xi = (U_\ell, C_\ell)$ where U_ℓ and C_ℓ are ℓ-bounded. A regular TPS of width ℓ is a TPS $\xi^* = (U^*_\ell, C_\ell)$ where U^*_ℓ is ℓ-regular and C_ℓ is ℓ-bounded.*

Fact 3.17 *α^*_i is production, and β_i and γ_i are consumptions.*

Fact 3.18 *For any width $\ell \in \mathbb{Z}^+$, the bounded TPS and the regular TPS are both covering TPSs.*

3.6.3 Properties of Activities

In this section, we discuss properties of activities based on restrictions on actions.

Given a directional vector set $U \in \mathbb{Q}^d$, we define the total order of the vectors as follows: (1) For any $\boldsymbol{u} \in U(i)$ and $\boldsymbol{v} \in U(j)$ where $i \neq j$, $\boldsymbol{u} < \boldsymbol{v}$ if and only if $i < j$; (2) For all $0 \leq i \leq d-1$, vectors in $U(i)$ are ordered lexicographically.

Definition 3.12 (Sorted activities) *A* sorted *activity M is an activity $(a_1, \cdots, a_k)$ such that (1) $\mathbf{u}(a_i)$ is in a directional vector set for all $1 \leq i \leq k$; (2) M does not include actions a_i, a_j that $\mathbf{u}(a_i) = \mathbf{u}(a_j)$; (3) a_i's in M are sorted in descending order of $\mathbf{u}(a_i)$'s.*

Given an activity M composed of directional vectors only, we construct a sorted activity $sort(M)$ by first replacing every pair of actions a_i, a_j that $\boldsymbol{u}(a_i) = \boldsymbol{u}(a_j)$ with $(\boldsymbol{u}(a_i), r(a_i) + r(a_j))$, and then sort actions a_i's in descending order of $\boldsymbol{u}(a_i)$'s. The following fact is immediate from the definition of activities and their costs.

Fact 3.19 *For any M composed of directional vectors only, $\boldsymbol{v}(M) = \boldsymbol{v}(sort(M))$ and $C(M) = C(sort(M))$.*

Proof. Assume $M = (a_1, \cdots, a_k)$. By definition, $\boldsymbol{v}(M) = \sum_i r_i \boldsymbol{u}_i$ and $C(M) = \sum_i r_i C(\boldsymbol{u}_i)$. Because vector addition and rational number addition are commutative and associative, it holds that $\boldsymbol{v}(M) = \boldsymbol{v}(sort(M))$ and $C(M) = C(sort(M))$. $\square$

To facilitate the analysis of activities constructed from directional vector sets, we often sort an activity M first and then partition the resulting sorted activity $sort(M)$ into sub-activities called *phases* such that each phase M_i only contains actions a's such that $\boldsymbol{u}(a) \in U(i)$. Thus $sort(M) = M_{d-1} \| \cdots \| M_0$.

Definition 3.13 (Simple activities) *A* simple *activity M is an activity $(a_1, \cdots, a_k)$ such that (1) $\boldsymbol{u}(a_i)$ is in a directional vector set for all $1 \leq i \leq k$; (2) For each i, $0 \leq i \leq d-1$, it is not the case that M_i has both production and consumption actions.*

Based on the ℓ-bounded vector sets, we further restrict sorted and simple activities and define normal activities.

Definition 3.14 (Normal activities) *A* normal *activity M is an activity $(a_1, \cdots, a_k)$ such that (1) there exists an ℓ-bounded vector set that includes $\mathbf{u}(a_i)$ for all $1 \le i \le k$; (2) M does not include any redemption γ_i for $i > 0$; (3) M is sorted; (4) M is simple.*

Normal activities have a clean structure where actions are sorted in order, not a single bucket contains both production and consumption actions, $\boldsymbol{\beta}_i$-type consumption can only appear for bucket $i > 0$, and $\boldsymbol{\gamma}_i$-type consumption can only appear for bucket $i = 0$.

It is possible that, given an activity M in a bounded (or regular) TPS, a normalized activity $norm(M)$ be constructed such that $\boldsymbol{v}(M) = \boldsymbol{v}(norm(M))$ and $C(M) \ge C(norm(M))$. This fact becomes very convenient in our analysis due to the nice structure of normal activities. In particular, if M is optimal and $\boldsymbol{v}(M) = \boldsymbol{p}$, then $norm(M)$ is also optimal and $\boldsymbol{v}(norm(M)) = \boldsymbol{p}$.

Lemma 3.20 *Given an activity M in a bounded (or regular) TPS, there exists a normal activity $norm(M)$ such that $\boldsymbol{v}(M) = \boldsymbol{v}(norm(M))$ and $C(M) \ge C(norm(M))$.*

We describe Algorithm 4 in next section that normalizes any activity in bounded TPSs. The proof of Lemma 3.20 follows from the correctness of this algorithm.

Activity Normalization

In this section, we describe the algorithm to normalize activities in bounded and regular TPSs. To facilitate the description and analysis of the algorithm, we first present several definitions and facts below. For any $I_x = (i_0, \cdots, i_k) \in \mathcal{I}_x$, define J_x as the prefix sum of I_x, i.e., $J_x[i] = \sum_{j=0}^{i} I_x[i]$ for all $0 \le i \le k$. For any $x \ge 1$, define $I_x^- = (i_0 - 1, i_1, \cdots, i_k)$ if $i_0 > 1$ and $I_x^- = (i_1, \cdots, i_k)$ otherwise.

Fact 3.21 *Consider the non-zero components in $\boldsymbol{\alpha}_{x,I_x}$ where $I_x = (i_0, \cdots, i_k) \in \mathcal{I}_x$. The 0-th such component is in bucket 0, and for any $1 \leq i \leq k+1$, the i-th such component is in bucket $J_x[i-1]$.*

Proof. Immediately follows the definition of production vector below.

$$\boldsymbol{\alpha}_{x,I_x} = (-i_0, \underbrace{0, \cdots, 0}_{i_0-1}, -i_1, \cdots, -i_k, \underbrace{0, \cdots, 0}_{i_k-1}, +\ell, \underbrace{0, \cdots, 0}_{d-1-x})$$

$\square$

Fact 3.22 *$I_x^- \in \mathcal{I}_{x-1}$. I_x^- is regular if and only if I_x is regular.*

Proof. Because $I_x = (i_0, \cdots, i_k)$ is an ℓ-bounded integer partition for x, we have $1 \leq i_j \leq \ell$ for all $0 \leq j \leq k$ and $\sum_{j=0}^{k} i_j = x$. By definition, $I_x^- = (i_0 - 1, i_1, \cdots, i_k)$ if $i_0 > 1$ and $I_x^- = (i_1, \cdots, i_k)$ otherwise. If $i_0 > 1$, we have $i_0 - 1 > 0$ and $\sum_{j=0}^{k} i_j - 1 = x - 1$. If $i_0 = 1$, we have $\sum_{j=1}^{k} i_j = x - 1$. In both cases, I_x^- is an ℓ-bounded integer partition for $x - 1$ and thus $I_x^- \in \mathcal{I}_{x-1}$.

When I_x is regular, it follows that $i_j = d$ for all $1 \leq j \leq k$, and $i_0 = d$ if $d \mid x$ and $i_0 = x \bmod d$ otherwise. Therefore, $I_x^- = ((x-1) \bmod d, d, \cdots, d)$ if $i_0 > 1$ and $I_x^- = (d, \cdots, d)$ otherwise. In both cases, I_x^- is regular. $\square$

Definition 3.15 (Replacement of redundant actions) *For any $x \geq 1$, $r > 0$ and $I_x \in \mathcal{I}_x$, define the* replacement *of a pair of actions $(\boldsymbol{\alpha}_{x,I_x}, r), (\boldsymbol{\beta}_x, r\ell)$ as an activity $R_{\alpha\beta}(x, I_x, r) = ((\boldsymbol{\alpha}_{x-1,I_x^-}, r), (\boldsymbol{\beta}_{J_x[k-1]}, ri_k), \cdots, (\boldsymbol{\beta}_{J_x[0]}, ri_1), (\boldsymbol{\gamma}_0, r))$, the* replacement *of an action $(\boldsymbol{\gamma}_x, r)$ as an activity $R_\gamma(x, r) = ((\boldsymbol{\beta}_x, r), (\boldsymbol{\beta}_{x-1}, r), \cdots, (\boldsymbol{\beta}_1, r), (\boldsymbol{\gamma}_0, r))$, and the* replacement *of a pair of actions $(\boldsymbol{\alpha}_{0,I_0}, r), (\boldsymbol{\gamma}_0, r\ell)$ as an activity $R_{\alpha\gamma} = (\boldsymbol{\gamma}_0, 0)$.*

Fact 3.23 *For all $x \geq 1$, the replacements $R_{\alpha\beta}(x, I_x, r)$, $R_\gamma(x, r)$, and $R_{\alpha\gamma}$ include only production and consumption vectors for buckets 0 to x.*

Computing vector sum and cost gives the following fact on redundant actions and their replacement.

Fact 3.24 *For any $x \geq 1$, $r > 0$, and $I_x \in \mathcal{I}_x$,*

$$\begin{aligned}
r\boldsymbol{\alpha}_{x,I_x} + r\ell\boldsymbol{\beta}_x &= \boldsymbol{v}(R_{\alpha\beta}(x, I_x, r)) \\
r \cdot C(\boldsymbol{\alpha}_{x,I_x}) + r\ell \cdot C(\boldsymbol{\beta}_x) &> C(R_{\alpha\beta}(x, I_x, r)) \\
r\boldsymbol{\gamma}_x &= \boldsymbol{v}(R_\gamma(x, r)) \\
r \cdot C(\boldsymbol{\gamma}_x) &= C(R_\gamma(x, r)) \\
r\boldsymbol{\alpha}_{0,I_0} + r\ell\boldsymbol{\gamma}_0 &= \boldsymbol{v}(R_{\alpha\gamma}) \\
r \cdot C(\boldsymbol{\alpha}_{0,I_0}) + r\ell \cdot C(\boldsymbol{\gamma}_0) &= C(R_{\alpha\gamma})
\end{aligned}$$

Now we are ready to present Algorithm 4 that normalizes activities in bounded TPSs. The basic idea is to do a scan from the top bucket to the bottom bucket, replacing possible violating actions in the bucket with their replacements. By Fact 3.23, such replacements do not include new production or consumption for higher indexed buckets, and thus cannot introduce new violating actions in higher buckets. Therefore, the top-down one-pass scan would simplify the entire activity.

Proof of Lemma 3.20: We first show that given any activity M in a bounded (or regular) TPS, Algorithm 4 computes a normal activity M':

- Assume the vectors of M are in the ℓ-bounded vector set U_ℓ. Then all the replacements used in M' are also in U_ℓ. In particular, by Fact 3.22, if all production vectors in M are regular, then all production vectors of the replacements in M' are also regular.

- The *for* loop from line 2 simplifies phase M'_x where x is from $d - 1$ down to 1.

Algorithm 4: Normalizing activities without increasing cost.

```
   Input: activity $M$ in a bounded TPS of dimension $d$ and width $\ell$
1  $M' \leftarrow sort(M)$
2  for $x \leftarrow d-1$ to $1$ do
3      if $M'_x$ contains $(\boldsymbol{\gamma}_x, r)$ for some $r > 0$ then
4          replace the action with $R_\gamma(x, r)$
5      while $M'_x$ contains $(\boldsymbol{\alpha}_{x,I}, r_1)$ and $(\boldsymbol{\beta}_x, r_2)$ for some $r_1, r_2 > 0$ do
6          $r \leftarrow \min(r_1, r_2/\ell)$
7          replace both actions with $R_{\alpha\beta}(x, I, r)$, $(\boldsymbol{\alpha}_{x,I}, r_1 - r)$, and $(\boldsymbol{\beta}_x, r_2 - r\ell)$
8  $M' \leftarrow sort(M')$
9  if $M'_0$ contains $(\boldsymbol{\alpha}_0, r_1)$ and $(\boldsymbol{\gamma}_0, r_2)$ for some $r_1, r_2 > 0$ then
10     $r \leftarrow \min(r_1, r_2/\ell)$
11     replace both actions with $R_{\alpha\gamma}$, $(\boldsymbol{\alpha}_0, r_1 - r)$, and $(\boldsymbol{\gamma}_0, r_2 - r\ell)$
12 Remove any action with 0 scalar from $M'$
13 $M' \leftarrow sort(M')$
   Output: $M'$
```

  - The *if* statement from line 3 in the *for* loop eliminates all $\boldsymbol{\gamma}_x$ vectors for phase M'_x without introducing productions or consumptions for buckets greater than x.
  - The *while* loop from line 5 in the *for* loop eliminates redundant $\boldsymbol{\alpha}_x$ or $\boldsymbol{\beta}_x$ vectors for phase M'_x without introducing productions or consumptions for buckets greater than x.

- The *if* statement from line 9 simplifies phase M'_0.
- M' is sorted before the final output.

Summing up the above observations, M' contains vectors in an ℓ-bounded vector set, and is sorted, simple, and does not include any γ_i for $i > 0$. Thus, by definition, M' is a normal activity.

By Fact 3.19, sorting an activity does not change the sum of the vectors and the total cost. By Fact 3.24, all the replacements do not change the sum of the vectors and do not increase the total cost. Therefore, we have $\boldsymbol{v}(M) = \boldsymbol{v}(M')$ and $C(M) \geq C(M')$. □

3.6.4 Reachability Results

In this section, we present results on reachability problems in TPSs.

Bounded Reachability

There is a situation that one wants to decide whether it is possible to reach a target state without ever hitting some lower bound condition in each step of the activity. For example, when a factory plans to conduct a series of productions, it needs to decide whether the demand for materials would ever exceeds the amount in stock in each step. This motivates the notion of *bounded reachability*. An activity $M = (a_1, \cdots, a_k)$ is $\boldsymbol{b}$-bounded if $\boldsymbol{p}_i \geq \boldsymbol{b}$ for all $1 \leq i \leq k-1$ in the sequence $\mathbf{0} \xrightarrow{a_1} \boldsymbol{p}_1 \rightarrow \cdots \rightarrow \boldsymbol{p}_{k-1} \xrightarrow{a_k} \boldsymbol{p}$. The bounded reachability problem is defined as follows.

Problem 3.11 The bounded reachability problem

1. **Instance:** A TPS (U, C) of dimension $d \in \mathbb{Z}^+$ and two vector $\boldsymbol{b}, \boldsymbol{p} \in \mathbb{Q}^d$.
2. **Objective:** Decide whether there exists a $\boldsymbol{b}$-bounded activity M such that $\mathbf{0} \xrightarrow{M} \boldsymbol{p}$.

For any two vectors $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{Q}^d$, define the minimum of the two vectors as $\boldsymbol{p} \wedge \boldsymbol{q}$ such that $(\boldsymbol{p} \wedge \boldsymbol{q})[i] = \min(\boldsymbol{p}[i], \boldsymbol{q}[i])$ for all $0 \leq i \leq d-1$.

Lemma 3.25 *For any covering TPS and any two vectors $\boldsymbol{b}$ and $\boldsymbol{p}$, if $\boldsymbol{b} \leq (\mathbf{0} \wedge \boldsymbol{p})$, then $\boldsymbol{p}$ is $\boldsymbol{b}$-bounded reachable from $\mathbf{0}$.*

Proof of Lemma 3.25: We show that Algorithm 5 computes M that satisfies the $\boldsymbol{b}$-bounded reachability for $\boldsymbol{b} = (\mathbf{0} \wedge \boldsymbol{p})$, and thus the case for $\boldsymbol{b} < (\mathbf{0} \wedge \boldsymbol{p})$ immediately follows. It is not hard to observe that $\mathbf{0} \xrightarrow{M} \boldsymbol{p}$ because for any i from $d-1$ down to 0, after iteration i of Algorithm 5, $\boldsymbol{p}[j] = 0$ for all $j \geq i$. As a result, $\boldsymbol{p} = \boldsymbol{v}(M)$. It remains to show that each intermediate vector is no smaller than $\mathbf{0} \wedge \boldsymbol{p}$. Let $M =$

Algorithm 5: Computing activities for bounded reachability in covering TPSs.

```
Input: A covering TPS $(U, C)$ of dimension $d$ and a vector $\boldsymbol{p}$
1  $h \leftarrow 0, t \leftarrow d-1$
2  for $i \leftarrow d-1$ to $0$ do
3      if $\boldsymbol{p}[i] > 0$ then
4          $\boldsymbol{u} \leftarrow$ the smallest production in $U(i)$
5          $r \leftarrow \boldsymbol{p}[i]/\boldsymbol{u}[i]$
6          $M_t \leftarrow (\boldsymbol{u}, r)$
7          $t \leftarrow t-1$
8      else
9          $\boldsymbol{u} \leftarrow$ the smallest consumption in $U(i)$
10         $r \leftarrow \boldsymbol{p}[i]/\boldsymbol{u}[i]$
11         $M_h \leftarrow (\boldsymbol{u}, r)$
12         $h \leftarrow h+1$
13     $\boldsymbol{p} \leftarrow \boldsymbol{p} - r\boldsymbol{u}$
Output: $M = M_0 \| \cdots \| M_{d-1}$
```

$((\boldsymbol{v}_0, r_0), \cdots, (\boldsymbol{v}_{d-1}, r_{d-1}))$. We prove the following invariant by reverse induction: for any bucket $0 \leq k \leq d-1$, the sequence $(\boldsymbol{v}_0[k], \boldsymbol{v}_1[k], \cdots, \boldsymbol{v}_{d-1}[k])$ satisfies that positive values, if any, come before negative values. For $0 \leq i \leq d-1$, let $\boldsymbol{u}_i$ be the vector chosen in iteration i by the algorithm. In iteration $d-1$, if $\boldsymbol{u}_{d-1}$ is a production, then it is placed at the end of the sequence M; otherwise it is placed at the beginning of the sequence M. Since $\boldsymbol{u}_{d-1}$ is the only vector in the sequence at this moment, the invariant holds after iteration $d-1$. Assume the invariant holds after iteration $i+1$. Now consider the inductive step in iteration i. Vector $\boldsymbol{u}_i$ is placed in M after all consumptions and before all productions from previous iterations. Let $\boldsymbol{u}$ and $\boldsymbol{u}'$ be any consumption and production from previous iterations, respectively. Then $\boldsymbol{u}[j] \geq 0, \boldsymbol{u}'[j] \leq 0$ for all $j \leq i$. Because $\boldsymbol{u}_i[j] = 0$ for all $j > i$, inserting $\boldsymbol{u}_i$ does not affect the invariant for all buckets $j > i$. If $\boldsymbol{u}_i$ is a consumption, then $\boldsymbol{u}_i[i] < 0$ and $\boldsymbol{u}_i[j] \geq 0$ for all $j < i$. If $\boldsymbol{u}_i$ is a production, then $\boldsymbol{u}_i[i] > 0$ and $\boldsymbol{u}_i[j] \leq 0$ for all $j < i$. The signs for buckets 0 to i in both cases are shown in the table below. Note that in both cases, the invariant holds for all these buckets $j \leq i$. Therefore, after inserting $\boldsymbol{u}_i$ in iteration i, the invariant still holds. Thus by induction we have shown that the invariant holds after all d iterations.

	consumption				production			
bucket	0	$\cdots$	$i-1$	i	0	$\cdots$	$i-1$	i
$\boldsymbol{u}$	$+$	$\cdots$	$+$	$+$	$+$	$\cdots$	$+$	$+$
$\boldsymbol{u}_i$	$+$	$\cdots$	$+$	$-$	$-$	$\cdots$	$-$	$+$
$\boldsymbol{u}'$	$-$	$\cdots$	$-$	$-$	$-$	$\cdots$	$-$	$-$

Based on the invariant, for any bucket $0 \leq k \leq d-1$, the prefix sum of the sequence $(\boldsymbol{v}_0[k], \boldsymbol{v}_1[k], \cdots, \boldsymbol{v}_{d-1}[k])$ is first weakly monotone increasing and then weakly monotone decreasing. Therefore, we conclude that in the sequence $\mathbf{0} \xrightarrow{\boldsymbol{v}_0} \boldsymbol{p}_0 \to \cdots \to \boldsymbol{p}_{d-2} \xrightarrow{\boldsymbol{v}_{d-1}} \boldsymbol{p}$, each intermediate vector $\boldsymbol{p}_i$ for $0 \leq i \leq d-2$ is no smaller than $\mathbf{0} \wedge \boldsymbol{p}$. □

Following Fact 3.18 and Lemma 3.25, we have the corollary below.

Corollary 3.26 *Let ξ and ξ^* be a pair of bounded and regular TPSs, and $\mathbf{b}, \mathbf{p}$ be two vectors. If $\mathbf{b} \leq (\mathbf{0} \wedge \mathbf{p})$, then $\mathbf{p}$ is $\mathbf{b}$-bounded reachable from $\mathbf{0}$ in ξ and ξ^*.*

Proof. By Fact 3.18, ξ and ξ^* are both covering TPSs. Then by Lemma 3.25, $\boldsymbol{p}$ is $\boldsymbol{b}$-bounded reachable from $\mathbf{0}$ in ξ and ξ^*. □

Min-Cost Reachability

Recall that Fact 3.10 states the weight of the sum of vectors is less than or equal to the sum of the vector weights. We show here that for reachability with simple activities in directional TPSs, this fact can be strengthened to imply equality.

Fact 3.27 *For any activities M_1 and M_2, if $M_1 + M_2$ is optimal, then both M_1 and M_2 are optimal.*

Proof. $M_1 + M_2$ is optimal and thus

$$\omega(\boldsymbol{v}(M_1 + M_2)) = C(M_1 + M_2) = C(M_1) + C(M_2)$$

Assume for contradiction that any of the two, say M_1, is not optimal. Then there must exist M_1' such that

$$\begin{aligned} \boldsymbol{v}(M_1') &= \boldsymbol{v}(M_1) \\ C(M_1') &= \omega(\boldsymbol{v}(M_1)) < C(M_1) \end{aligned}$$

Then replacing M_1 with M_1' gives

$$\begin{aligned} \boldsymbol{v}(M_1' + M_2) &= \boldsymbol{v}(M_1 + M_2) \\ C(M_1' + M_2) &< \omega(\boldsymbol{v}(M_1' + M_2)) \end{aligned}$$

This is a contradiction because $\omega(\boldsymbol{v}(M_1'+M_2))$ is the minimum cost to reach $\boldsymbol{v}(M_1'+M_2)$. Therefore, both of M_1 and M_2 must be optimal. $\square$

Given an activity $M = ((\boldsymbol{u}_1, r_1), \cdots, (\boldsymbol{u}_k, r_k))$ and a vector $\boldsymbol{x} = (x_1, \cdots, x_k)$ such that $\boldsymbol{x} \geq \boldsymbol{0}$, define the *dot product* of a vector and an activity $(\boldsymbol{x} \cdot M) = ((\boldsymbol{u}_1, r_1 x_1), \cdots, (\boldsymbol{u}_k, r_k x_k))$. Therefore, $(\boldsymbol{x} \cdot M)$ is also an activity. We have the following facts about an activity and its dot product with a vector.

Fact 3.28 *Let M be an optimal activity and $\boldsymbol{0} \leq \boldsymbol{x} \leq \boldsymbol{1}$. Then $(\boldsymbol{x} \cdot M)$ is also an optimal activity.*

Proof. Let $\boldsymbol{y} = \boldsymbol{1} - \boldsymbol{x}$. Then $M = \boldsymbol{x} \cdot M + \boldsymbol{y} \cdot M$. By Fact 3.27, because M is optimal, both $(\boldsymbol{x} \cdot M)$ and $(\boldsymbol{y} \cdot M)$ are optimal. $\square$

Fact 3.29 *Let M be an activity and $\boldsymbol{x}, \boldsymbol{y} \geq \mathbf{0}$. Then $C((\boldsymbol{x}+\boldsymbol{y})\cdot M) = C(\boldsymbol{x}\cdot M)+C(\boldsymbol{y}\cdot M)$.*

Proof. Let $\boldsymbol{x} = (x_1, \cdots, x_k)$, $\boldsymbol{y} = (y_1, \cdots, y_k)$, $M = ((\boldsymbol{u}_1, r_1), \cdots, (\boldsymbol{u}_k, r_k))$. By definition of dot product and the cost of an activity,

$$\begin{aligned} C((\boldsymbol{x}+\boldsymbol{y})\cdot M) &= \sum_{i=1}^{k}(x_i + y + i) r_i C(\boldsymbol{u}_i) \\ &= \sum_{i=1}^{k} x_i r_i C(\boldsymbol{u}_i) + \sum_{i=1}^{k} y_i r_i C(\boldsymbol{u}_i) \\ &= C(\boldsymbol{x}\cdot M) + C(\boldsymbol{y}\cdot M) \end{aligned}$$

□

Fact 3.30 *Let M be a simple activity and $\boldsymbol{x} \geq \mathbf{0}$. Then $(\boldsymbol{x}\cdot M)$ is also a simple activity.*

Proof. M is simple. $(\boldsymbol{x}\cdot M)$ only scales the scalars of actions in M, but does not change the vectors. Therefore, $(\boldsymbol{x}\cdot M)$ is also simple. □

Lemma 3.31 *Let $\boldsymbol{p}, \boldsymbol{q} \geq \mathbf{0}$ (or $\leq \mathbf{0}$) and M be a simple activity such that $\boldsymbol{v}(M) = \boldsymbol{p}+\boldsymbol{q}$. Then there exist vectors $\boldsymbol{x}, \boldsymbol{y}$ such that $\mathbf{0} \leq \boldsymbol{x} \leq \mathbf{1}$, $\boldsymbol{y} = 1 - \boldsymbol{x}$, $\boldsymbol{v}(\boldsymbol{x}\cdot M) = \boldsymbol{p}$, and $\boldsymbol{v}(\boldsymbol{y}\cdot M) = \boldsymbol{q}$.*

Algorithm 6: Constructing activities to reach $\boldsymbol{p}$ and $\boldsymbol{q}$.

```
  Input: $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{Q}^d$, a simple activity $M$ such that $\boldsymbol{v}(M) = \boldsymbol{p}+\boldsymbol{q}$
1 $M \leftarrow sort(M)$
2 for $i \leftarrow d-1$ to $0$ do
3     $x_i \leftarrow \boldsymbol{p}[i]/(\boldsymbol{p}[i]+\boldsymbol{q}[i])$
4     $y_i \leftarrow 1 - x_i$
5     $M_i^p \leftarrow x_i M_i$
6     $M_i^q \leftarrow y_i M_i$
7     $\boldsymbol{p} \leftarrow \boldsymbol{p} - x_i M_i$
8     $\boldsymbol{q} \leftarrow \boldsymbol{q} - y_i M_i$
  Output: $M^p = M_0^p \| \cdots \| M_{d-1}^p$ and $M^q = M_0^q \| \cdots \| M_{d-1}^q$
```

Proof of Lemma 3.31: We show that Algorithm 6 computes $\boldsymbol{x}$ and $\boldsymbol{y}$, such that $M^p = (\boldsymbol{x} \cdot M)$ and it reaches $\boldsymbol{p}$, and $M^q = (\boldsymbol{y} \cdot M)$ and it reaches $\boldsymbol{q}$. Note that $\boldsymbol{p}$ and $\boldsymbol{q}$ have the same sign and M is simple, then $x_i = \boldsymbol{p}[i]/(\boldsymbol{p}[i] + \boldsymbol{q}[i])$ is always in the range $[0, 1]$ and so does $y_i = 1 - x_i$. Therefore, $\mathbf{0} \leq \boldsymbol{x} \leq \mathbf{1}$ and $\boldsymbol{y} = \mathbf{1} - \boldsymbol{x}$. To show that $\boldsymbol{v}(M^p) = \boldsymbol{p}$ and $\boldsymbol{v}(M^q) = \boldsymbol{q}$, it is sufficient to observe that for any i from $d - 1$ down to 0, after iteration i of the algorithm, $\boldsymbol{p}[j] = \boldsymbol{q}[j] = 0$ for all $j \geq i$ and $\boldsymbol{p}[j], \boldsymbol{q}[j]$ have the same sign for all $j < i$. □

Now we are ready to strengthen Fact 3.10 to imply equality. The result is stated as follows.

Lemma 3.32 *Let $\mathbf{p}, \mathbf{q} \geq \mathbf{0}$ (or $\leq \mathbf{0}$) and M be a simple and optimal activity such that $\boldsymbol{v}(M) = \boldsymbol{p} + \boldsymbol{q}$. Then $\omega(\boldsymbol{p} + \boldsymbol{q}) = \omega(\boldsymbol{p}) + \omega(\boldsymbol{q})$.*

Proof. By Lemma 3.31, there exist $\boldsymbol{x} + \boldsymbol{y} = \mathbf{1}$ such that $M_1 = (\boldsymbol{x} \cdot M)$, $M_2 = (\boldsymbol{y} \cdot M)$, and M_1 and M_2 reach $\boldsymbol{p}$ and $\boldsymbol{q}$, respectively. By Fact 3.27, because M is optimal, M_1 and M_2 are both optimal. Therefore, we have

$$\begin{aligned} C(M) &= \omega(\boldsymbol{p} + \boldsymbol{q}) \\ C(M_1) &= \omega(\boldsymbol{p}) \\ C(M_2) &= \omega(\boldsymbol{q}) \end{aligned}$$

By Fact 3.29,

$$C(M) = C(M_1) + C(M_2)$$

As a result,

$$\omega(\boldsymbol{p} + \boldsymbol{q}) = \omega(\boldsymbol{p}) + \omega(\boldsymbol{p})$$

□

Following Fact 3.9 and Lemma 3.32, we have the lemma below to indicate the linearity of minimum reachability cost with simple activities in directional TPSs.

Lemma 3.33 *For any directional TPS, let $\boldsymbol{p}, \boldsymbol{q} \geq \mathbf{0}$ (or $\leq \mathbf{0}$), $r_1, r_2 \geq 0$, and M be a simple and optimal activity such that $\boldsymbol{v}(M) = r_1\boldsymbol{p} + r_2\boldsymbol{q}$. Then $\omega(r_1\boldsymbol{p} + r_2\boldsymbol{q}) = r_1\omega(\boldsymbol{p}) + r_2\omega(\boldsymbol{q})$.*

Proof. By Lemma 3.32, we have

$$\omega(r_1\boldsymbol{p} + r_2\boldsymbol{q}) = \omega(r_1\boldsymbol{p}) + \omega(\boldsymbol{r_2 q}) \tag{3.8}$$

By Fact 3.9, we have

$$\omega(r_1\boldsymbol{p}) = r_1\omega(\boldsymbol{p}) \tag{3.9}$$

$$\omega(r_2\boldsymbol{p}) = r_2\omega(\boldsymbol{p}) \tag{3.10}$$

Combining (3.8), (3.9), and (3.10) gives

$$\omega(r_1\boldsymbol{p} + r_2\boldsymbol{q}) = r_1\omega(\boldsymbol{p}) + r_2\omega(\boldsymbol{q})$$

□

For bounded TPSs, this linearity property easily follows due to the following lemma as a result of Lemma 3.20 and Corollary 3.26.

Lemma 3.34 *Let ξ and ξ^* be a pair of bounded and regular TPSs, and $\mathbf{p}$ be a vector. Then there exist normal and optimal activities M in ξ and M^* in ξ^*, such that $\boldsymbol{v}(M) = \boldsymbol{v}(M^*) = \boldsymbol{p}$.*

Proof. By Corollary 3.26, $\boldsymbol{p}$ is reachable from $\mathbf{0}$ in ξ and ξ^*. Assume $\mathbf{0} \xrightarrow{M_1} \boldsymbol{p}$ and $C(M_1) = \omega(\boldsymbol{p})$ in ξ, and $\mathbf{0} \xrightarrow{M_2} \boldsymbol{p}$ and $C(M_2) = \omega(\boldsymbol{p})$ in ξ^*. Let $M = norm(M_1)$ and

$M^* = norm(M_2)$. Then by Lemma 3.20, $\mathbf{0} \xrightarrow{M} \boldsymbol{p}$ and $C(M) \leq C(M_1) = \omega(\boldsymbol{p})$ in ξ, and $\mathbf{0} \xrightarrow{M^*} \boldsymbol{p}$ and $C(M^*) \leq C(M_2) = \omega(\boldsymbol{p})$ in ξ^*. Therefore, M and M^* are normal and optimal activities to reach $\boldsymbol{p}$ in ξ and ξ^*, respectively. $\square$

Then following Lemma 3.33 and Lemma 3.34, we have

Corollary 3.35 *Let ξ and ξ^* be a pair of bounded and regular TPSs, $\boldsymbol{p}, \boldsymbol{q} \geq \mathbf{0}$ (or $\leq \mathbf{0}$), $r_1, r_2 \geq 0$. Then $\omega(r_1\boldsymbol{p} + r_2\boldsymbol{q}) = r_1\omega(\boldsymbol{p}) + r_2\omega(\boldsymbol{q})$ in both ξ and ξ^*.*

Proof. By Lemma 3.34, there exist normal and optimal activities M in ξ and M^* in ξ^*, such that $\boldsymbol{v}(M) = \boldsymbol{v}(M^*) = r_1\boldsymbol{p} + r_2\boldsymbol{q}$. Note bounded and regular TPSs are directional by definition, and normal strategies are simple by definition. Then it follows from Lemma 3.33 that $\omega(r_1\boldsymbol{p} + r_2\boldsymbol{q}) = r_1\omega(\boldsymbol{p}) + r_2\omega(\boldsymbol{q})$ in ξ and ξ^*. $\square$

3.6.5 Minimum Token Production Cost

To motivate the notation of *token production cost*, imagine the following activity in a factory that it first produces some products, potentially with the consumption of some materials, and then clears out the stock of the produced products and restocks the consumed materials. After such an activity, the stock of everything goes back to the state it used to be before the activity. We call the cost of such an activity the token production cost. Because there might be many such activities with different costs, we are interested in determining the minimum token production cost.

The minimum token production cost problem is a special case of the min-cost 1-stop reachability problem. Recall the min-cost 1-stop reachability problem is that given a TPS, a finite set of vectors V, and a vector $\boldsymbol{p}$, to determine the minimum cost to reach $\boldsymbol{p}$ from $\mathbf{0}$ via some intermediate vector $\boldsymbol{q} \in V$. If we set the condition that V is the set of all productions for the bucket representing the product that the factory wants to produce, and $\boldsymbol{p} = \mathbf{0}$, then any activity that satisfies 1-stop reachability corresponds to a way of token production. Therefore, the minimum cost for 1-stop reachability is equal to the minimum token production cost.

We investigate the minimum token production cost (TPC) problem in bounded TPSs. Given $d, \ell \in \mathbb{Z}^+$, the bounded TPS ξ of dimension d and width ℓ is uniquely defined. Let $\mathcal{M}(d, \ell)$ be the set of all possible activities in ξ that first reach some production vector for bucket $d-1$ and then go back to $\mathbf{0}$. We call these activities as the *token production activities*. Formally,

$$\mathcal{M}(d,\ell) = \{M : M = M_1 \| M_2, \mathbf{0} \xrightarrow{M_1} \boldsymbol{q} \xrightarrow{M_2} \mathbf{0}, \\ \boldsymbol{q} \in V, V = \{\boldsymbol{\alpha}_{d-1,I} : I \in \mathcal{I}_{d-1}\}\}$$

Then for an activity $M \in \mathcal{M}(d, \ell)$, $C(M)$ is called the token production cost of ξ via M. The minimum of $C(M)$ over all $M \in \mathcal{M}(d, \ell)$ is the minimum token production cost of ξ.

Problem 3.12 The Minimum TPC Problem in bounded TPSs

1. **Instance:** $d, \ell \in \mathbb{Z}^+$.
2. **Objective:** Determine $\min\{C(M) : M \in \mathcal{M}(d, \ell)\}$.

3.6.6 Embedding Witness Graphs in TPS Activities

Given witness graphs in $\mathcal{G}(d, \ell)$, we construct activities in bounded TPSs and show that the minimum production cost provides a lower bound for the minimum graph weight. The basic idea is that having a vertex v in a witness graph corresponds to having certain actions in bucket $d_{\mathrm{i}}(v)$ and below, which create $d_{\mathrm{o}}(v)$ units of supply in buckets according to v's outgoing edges and $d_{\mathrm{i}}(v)$ units of demand in buckets according to v's incoming edges. In the whole graph, the total weight of outgoing edges equals that of incoming edges.

Therefore, the corresponding activity creates supply that meets the demand, satisfying the final state $\mathbf{0}$ for token productions.

Definition 3.16 (Activity from witness graph) *For any $d, \ell \in \mathbb{Z}^+$ and any $G = (V, E, \phi, w) \in \mathcal{G}(d, \ell)$, we construct an activity M^G consisting of vectors in $U_\ell \subset \mathbb{Z}^{d+1}$ as follows. Sort vertices $v \in V$ in descending order of $\mathrm{d_i}(v)$'s. M^G starts as empty. For each vertex $v \in V$, suppose v has x incoming edges $c_0, \cdots, c_{x-1}$ and y outgoing edges $e_0, \cdots, e_{y-1}$ listed in the order compatible with ϕ. For each $v \in V$ do:*

1. *Append $\left(\boldsymbol{\alpha}_{\mathrm{d_i}(v),I}, 1\right)$ to M^G, where $I = (w(c_0), \cdots, w(c_{x-1}))$.*
2. *Append $\left(\boldsymbol{\gamma}_{\mathrm{d_i}(v)}, \ell - \mathrm{d_o}(v)\right)$ to M^G.*
3. *For v's each outgoing edge e_j where $0 \le j \le y-1$ do: if $\mathrm{d_i}(v) \ge \eta(e_j) + 1$, then append $\left(\boldsymbol{\beta}_{\mathrm{d_i}(v)}, w(e_j)\right)$, $\left(\boldsymbol{\beta}_{\mathrm{d_i}(v)-1}, w(e_j)\right)$, $\cdots$, $(\boldsymbol{\beta}_{\eta(e_j)+1}, w(e_j))$ to M^G; otherwise do nothing.*

For example, if we are given the witness graph G shown in Figure 3.4, we construct an activity M^G as follows. For vertex s_2, M^G includes $(\boldsymbol{\alpha}_{5,(1,2,2)}, 1)$ in the first step, $(\boldsymbol{\gamma}_5, 0)$ in the second step, $(\boldsymbol{\beta}_5, 2)$, $(\boldsymbol{\beta}_4, 2)$ for outgoing edge ④ and $(\boldsymbol{\beta}_5, 1)$, $(\boldsymbol{\beta}_4, 1)$, $(\boldsymbol{\beta}_3, 1)$, $(\boldsymbol{\beta}_2, 1)$, $(\boldsymbol{\beta}_1, 1)$ for outgoing edge ① in the third step. For vertex s_1, M^G includes $(\boldsymbol{\alpha}_{1,(1)}, 1)$ in the first step, $(\boldsymbol{\gamma}_1, 0)$ in the second step, and $(\boldsymbol{\beta}_1, 1)$ for outgoing edge ② in the third step. The actions are summarized in Table 3.1. It can be easily observed that $\boldsymbol{v}(M^G) = \mathbf{0}$ and $C(M^G) = 6$.

We show that in general if $G \in \mathcal{G}(d, \ell)$, then $M^G \in \mathcal{M}(d+1, \ell)$ and $C(M^G) = W(G)$, and thus a lower bound for minimum production cost in $\mathcal{M}(d+1, \ell)$ provides a lower bound on the minimum weight of witness graphs in $\mathcal{G}(d, \ell)$.

Lemma 3.36 *For any $d, \ell \in \mathbb{Z}^+$ and any $G \in \mathcal{G}(d, \ell)$, let M^G be the activity constructed from G following Definition 3.16. Then $M^G \in \mathcal{M}(d+1, \ell)$ and $C(M^G) = W(G)$.*

Table 3.1: Construction of an activity from a witness graph

vertex	s_2							s_1						
bucket		0	1	2	3	4	5		0	1	2	3	4	5
action	$\boldsymbol{\alpha}_{5,(1,2,2)}$	-1	-2	0	-2	0	3	$\boldsymbol{\alpha}_{1,(1)}$	-1	3	0	0	0	0
	$2\sum_{i=4}^{5}\boldsymbol{\beta}_i$	0	0	0	2	0	-2	$\boldsymbol{\beta}_1$	1	-1	0	0	0	0
	$\sum_{i=1}^{5}\boldsymbol{\beta}_i$	1	0	0	0	0	-1							
sum		0	-2	0	0	0	0		0	2	0	0	0	0

Proof. Note that buckets of an activity in $\mathcal{M}(d+1,\ell)$ are indexed from 0 to d. We first show that M^G reaches $\boldsymbol{q} = \boldsymbol{\alpha}_{d,I}$ for some $I \in \mathcal{I}_d$. Let v be the first vertex picked in the construction. Then $d_{\mathrm{i}}(v) = d$ because $d_{\mathrm{i}}^G = d$ and vertices are sorted in descending order of their in-degree. Let $c_0, \cdots, c_{x-1}$ be the incoming edges of v. Then $\sum_{j=0}^{x-1} w(c_j) = d$ and each $0 \leq w(c_j) \leq \ell$. During the first iteration in the construction, M^G includes $(\boldsymbol{\alpha}_{d,I}, 1)$ as the first action, where $I = (w(c_0), \cdots, w(c_{x-1})) \in \mathcal{I}_d$. Therefore, $\mathbf{0} \xrightarrow{(\boldsymbol{\alpha}_{d,I},1)} \boldsymbol{\alpha_{d,I}}$.

Next we show that $\boldsymbol{v}(M^G) = \mathbf{0}$ and thus $M^G - (\boldsymbol{\alpha}_{d,I}, 1)$ cancels $\boldsymbol{\alpha_{d,I}}$. Write $M^G = ((\boldsymbol{a}_0, r_0), \cdots, (\boldsymbol{a}_i, r_i), \cdots)$. Let $\boldsymbol{p}[i]_v$ represent the amount that actions corresponding to vertex v add to bucket i. We complete the proof by reverse induction on i. The base case is when $i = d$. Only vertices with in-degree d would affect the value of bucket d. For each such vertex v, M^G adds actions in three steps: step 1 adds ℓ to bucket d, step 2 subtracts $\ell - d_{\mathrm{o}}(v)$ from bucket d, and step 3 subtracts $\sum_j w(e_j) = d_{\mathrm{o}}(v)$ from bucket d. Thus,

$$\boldsymbol{p}[d]_v = \ell - (\ell - d_{\mathrm{o}}(v)) - d_{\mathrm{o}}(v) = 0$$

Summing over all vertices gives

$$\sum_{v \in V} \boldsymbol{p}[d]_v = \sum_{v: d_{\mathrm{i}}(v) = d} \boldsymbol{p}[d]_v = 0$$

Assume for all $i > 0$,

$$\sum_{v\in V} \boldsymbol{p}[i]_v = 0 \tag{3.11}$$

Now consider the inductive step when $i = 0$. Consider the graph G as a whole and let $\boldsymbol{p}_v = \sum_{j=0}^{d} \boldsymbol{p}[j]_v$. For each $v \in V$, M^G adds actions in three steps: step 1 adds $\ell - d_{\mathrm{i}}(v)$ to $\boldsymbol{p}_v$, step 2 subtracts $\ell - d_{\mathrm{o}}(v)$ from $\boldsymbol{p}_v$, and step 3 does not change $\boldsymbol{p}_v$. Thus,

$$\begin{aligned} \boldsymbol{p}_v &= (\ell - d_{\mathrm{i}}(v)) - (\ell - d_{\mathrm{o}}(v)) \\ &= d_{\mathrm{o}}(v) - d_{\mathrm{i}}(v) \end{aligned}$$

Summing over all vertices gives

$$\sum_{v\in V} \boldsymbol{p}_v = \sum_{v\in V} (d_{\mathrm{o}}(v) - d_{\mathrm{i}}(v)) = 0 \tag{3.12}$$

Combining (3.11) and (3.12) gives

$$\sum_{v\in V} \boldsymbol{p}[0]_v = \sum_{v\in V} \boldsymbol{p}_v - \sum_{i>0} \sum_{v\in V} \boldsymbol{p}[i]_v = 0$$

As a result, for all $0 \le i \le d$, M^G adds 0 to bucket i and thus $\boldsymbol{v}(M^G) = \mathbf{0}$. Therefore, $\mathbf{0} \xrightarrow{(\boldsymbol{\alpha}_{d,I},1)} \boldsymbol{\alpha}_{d,I} \xrightarrow{M-(\boldsymbol{\alpha}_{d,I},1)} \mathbf{0}$ and thus $M^G \in \mathcal{M}(d+1,\ell)$.

It remains to show that $C(M^G) = W(G)$. For any $v \in V$, M^G adds moves in three steps: step 1 costs ℓ, step 2 costs $-(\ell - d_{\mathrm{o}}(v))$, and step 3 costs 0. Summing over all $v \in V$ gives

$$\begin{aligned} C(M^G) &= \sum_{v\in V} (\ell - (\ell - d_{\mathrm{o}}(v))) \\ &= \sum_{v\in V} d_{\mathrm{o}}(v) = \sum_{v\in V} d_{\mathrm{o}}(v) \\ &= W(G) \end{aligned}$$

□

Now we establish the following lemma that the minimum production cost provides a lower bound for the minimum graph weight problem.

Lemma 3.37 *For any* $d, \ell \in \mathbb{Z}^+$, $\min_{M \in \mathcal{M}(d+1,\ell)} C(M) \leq \min_{G \in \mathcal{G}(d,\ell)} W(G)$.

Proof. Let $G \in \mathcal{G}(d, \ell)$ and M^G be the activity constructed from G. By Lemma 3.36, $M^G \in \mathcal{M}(d+1, \ell)$ and $C(M^G) = W(G)$. Let $\mathcal{M}' = \{M^G : G \in \mathcal{G}(d, \ell)\}$. Then $\mathcal{M}' \subseteq \mathcal{M}(d+1, \ell)$. It follows that

$$\min_{M \in \mathcal{M}(d+1,\ell)} C(M) \leq \min_{M \in \mathcal{M}'} C(M) = \min_{G \in \mathcal{G}(d,\ell)} W(G)$$

□

3.7 Bounds on Token Production Cost and Competitive Ratio

In this section, we first solve minimum token production cost problem, and then derive lower bounds for minimum graph weight problem and minimum total load problem. As a consequence of such results, we finally prove Theorem 3.3.

3.7.1 Solving Minimum Token Production Cost

Following Lemma 3.15, the minimum TPC problem can be formulated by a number of linear programming problems that equals the number of ℓ-bounded integer partitions for $d-1$. This formulation requires exponential number of decision variables in terms of d and ℓ. We present here a combinatorial analysis that needs only $3d-1$ vectors to construct an optimal activity.

Lemma 3.38 *Let ξ and ξ^* be a pair of bounded and regular TPSs of dimension d and width ℓ, where $d, \ell \in \mathbb{Z}^+$. Then $\omega(\boldsymbol{\alpha}_{d-1}) = \ell$ in both ξ and ξ^*.*

Proof. It is straightforward that $\mathbf{0} \overset{(\boldsymbol{\alpha}_{d-1},1)}{\longrightarrow} \boldsymbol{\alpha}_{d-1}$ and thus

$$\omega(\boldsymbol{\alpha}_{d-1}) \leq C(\boldsymbol{\alpha}_{d-1}) = \ell \tag{3.13}$$

We show that optimal activities have cost no less than ℓ by a simple counting argument. The sum of the components of $\boldsymbol{\alpha}_{d-1}$ is $\ell - (d-1)$. By Lemma 3.34, there exists a normal activity M such that $\boldsymbol{v}(M) = \boldsymbol{\alpha}_{d-1}$ and $C(M) = \omega(\boldsymbol{\alpha}_{d-1})$. Assume for sake of contradiction that

$$C(M) < \ell$$

Because M is normal, it is simple and thus M_{d-1} only has productions. Therefore, the sum of scalars in M_{d-1} is 1. Then the sum of the components of vectors in M_{d-1} is thus $\ell - (d-1)$, exactly the same as that in $\boldsymbol{\alpha}_{d-1}$. Also,

$$C(M_{d-1}) = \ell$$

Therefore, if $\boldsymbol{v}(M) = \boldsymbol{\alpha}_{d-1}$ and $C(M) < \ell$, it must be the case that the sum of the components of vectors in $(M_0 \| \cdots \| M_{d-2})$ is 0 and

$$C(M_0 \| \cdots \| M_{d-2}) < 0 \tag{3.14}$$

The sum of components and the cost of each $\boldsymbol{\alpha}_i$, $\boldsymbol{\beta}_i$, and $\boldsymbol{\gamma}_i$ vector are summarized in the table below.

	sum of components	cost
$\boldsymbol{\alpha}_i$	$\leq \ell$	ℓ
$\boldsymbol{\beta}_i$	0	0
$\boldsymbol{\gamma}_i$	-1	-1

Let the sum of scalars of $\boldsymbol{\alpha}_i$, $\boldsymbol{\beta}_i$, and $\boldsymbol{\gamma}_i$ vectors in $(M_0\| \cdots \|M_{d-2})$ be x, y, and z, respectively. Because the sum of the components of vectors in $(M_0\| \cdots \|M_{d-2})$ is 0, it must be true that

$$\ell \cdot x + 0 \cdot y + (-1) \cdot z \geq 0$$

This implies that

$$C(M_0\| \cdots \|M_{d-2}) \geq 0 \tag{3.15}$$

(3.15) contradicts with (3.14) and thus

$$\omega(\boldsymbol{\alpha}_{d-1}) = C(M) \geq \ell \tag{3.16}$$

Finally, combining (3.13) and (3.16) gives

$$\omega(\boldsymbol{\alpha}_{d-1}) = \ell$$

□

By Fact 3.13, an optimal activity M such that $\boldsymbol{\alpha}_{d-1} \xrightarrow{M} \mathbf{0}$ is also optimal for $\mathbf{0} \xrightarrow{M} -\boldsymbol{\alpha}_{d-1}$. Therefore, $C(M) = \omega(-\boldsymbol{\alpha}_{d-1})$. With regard to this idea of cancelling a vector to reach zero, we have the following definitions. Given the dimension d and the width ℓ, consider a pair of the bounded TPS ξ and the regular TPS ξ^*. For any $\boldsymbol{p}$, define its *cancellation cost* $ccost(\boldsymbol{p})$ to be $\omega(-\boldsymbol{p})$ in ξ, and its *regular cancellation cost* $rcost(\boldsymbol{p})$ to be $\omega(-\boldsymbol{p})$ in ξ^*. We analyze the cost to cancel a vector in both bounded and regular TPSs, and show $rcost(\boldsymbol{\alpha}^*_{d-1}) \leq rcost(\boldsymbol{\alpha}_{d-1})$ and $ccost(\boldsymbol{\alpha}^*_{d-1}) = rcost(\boldsymbol{\alpha}^*_{d-1})$.

We compute the regular cancellation cost for a single negative token and show this cost is monotone increasing in its bucket index.

Lemma 3.39 *For any $0 \leq x \leq d-1$, let $\boldsymbol{p}_x$ be the vector such that $\boldsymbol{p}_x[x] = -1$ and $\boldsymbol{p}_x[j] = 0$ for all $j \neq x$. Then $rcost(\boldsymbol{p}_x) = 2^{\lfloor x/\ell \rfloor}(1 + (x \bmod \ell)/\ell)$ and $rcost(\boldsymbol{p}_x)$ is monotone increasing in x.*

Proof. Let $q_i = \lfloor i/\ell \rfloor$ and $r_i = i \bmod \ell$. We show $rcost(\boldsymbol{p}_x) = 2^{q_x}(1 + r_x/\ell)$ by induction on i. The base case is when $i = 0$. In this case $\mathcal{I}_0 = \{I^*_0\}$. It is straightforward that $rcost(\boldsymbol{p}_0) = -\boldsymbol{p}_0[0] = 1$. Note $q_0 = 0, r_0 = 0$ and thus $rcost(\boldsymbol{p}_0) = 2^{q_0}(1 + r_0/\ell)$.

Assume that $rcost(\boldsymbol{p}_i) = 2^{q_i}(1 + r_i/\ell)$ holds for all $i < x$. Now consider the inductive step when $i = x$. By Lemma 3.34, there exists a normal and optimal activity M that cancels $\boldsymbol{p}_x$. Therefore, M is simple and thus $M_x = (\boldsymbol{\alpha}^*_x, 1/\ell)$ and

$$C(M_x) = 1 \tag{3.17}$$

Then after phase M_x we have

$$\begin{aligned} \boldsymbol{p} &= \boldsymbol{p}_x + \boldsymbol{v}(M_x) \\ &= (-r_x/\ell, \underbrace{0, \cdots, 0}_{r_x-1}, -1, \underbrace{0, \cdots, 0}_{\ell-1}, \cdots, -1, 0, \cdots, 0), \end{aligned}$$

where $\boldsymbol{p}[0] = -r_x/\ell, \boldsymbol{p}[r_x + j\ell] = -1$ for all $0 \leq j \leq q_x - 1$. Thus,

$$\boldsymbol{p} = \frac{r_x}{\ell}\boldsymbol{p}_0 + \sum_{j=0}^{q_x-1} \boldsymbol{p}_{r_x+j\ell}$$

By Corollary 3.35,

$$rcost(\boldsymbol{p}) = \frac{r_x}{\ell} \cdot rcost(p_0) + \sum_{j=0}^{q_x-1} rcost(\boldsymbol{p}_{r_x+j\ell})$$

Then by induction hypothesis,

$$\begin{aligned} rcost(\boldsymbol{p}) &= \frac{r_x}{\ell} + \sum_{j=0}^{q_x-1} 2^j \left(1 + \frac{r_x}{\ell}\right) \\ &= 2^{q_x}\left(1 + \frac{r_x}{\ell}\right) - 1 \end{aligned} \tag{3.18}$$

Combining (3.17) and (3.18) gives

$$rcost(\boldsymbol{p}_x) = C(M_x) + rcost(\boldsymbol{p}) = 2^{q_x}\left(1 + \frac{r_x}{\ell}\right)$$

To show $rcost(\boldsymbol{p}_x)$ is monotone increasing in x, assume $i < j$ and let $q_i = \lfloor i/\ell \rfloor$, $r_i = i \bmod \ell$, $q_j = \lfloor j/\ell \rfloor$, and $r_j = j \bmod \ell$. We want to show $rcost(\boldsymbol{p}_i) < rcost(\boldsymbol{p}_j)$. If $q_i = q_j$, then $r_i < r_j$. In this case,

$$\begin{aligned} & rcost(\boldsymbol{p}_j) - rcost(\boldsymbol{p}_i) \\ = \; & 2^{q_j}\left(1 + \frac{r_j}{\ell}\right) - 2^{q_i}\left(1 + \frac{r_i}{\ell}\right) \\ = \; & 2^{q_i}\left(\frac{r_j - r_i}{\ell}\right) > 0 \end{aligned}$$

If $q_i < q_j$, then $r_i - r_j \leq \ell - 1$. In this case,

$$\begin{aligned}
& rcost(\boldsymbol{p}_j) - rcost(\boldsymbol{p}_i) \\
= \ & 2^{q_j}\left(1 + \frac{r_j}{\ell}\right) - 2^{q_i}\left(1 + \frac{r_i}{\ell}\right) \\
= \ & 2^{q_i}\left[2^{q_j - q_i}\left(1 + \frac{r_j}{\ell}\right) - \left(1 + \frac{r_i}{\ell}\right)\right] \\
\geq \ & 2^{q_i}\left[2\left(1 + \frac{r_j}{\ell}\right) - \left(1 + \frac{r_i}{\ell}\right)\right] \\
\geq \ & 2^{q_i}\left(\frac{r_j + 1}{\ell}\right) > 0
\end{aligned}$$

Therefore for any $i < j$, we have shown that $rcost(\boldsymbol{p}_i) < rcost(\boldsymbol{p}_j)$. Thus, $rcost(\boldsymbol{p}_x)$ is monotone increasing in x. $\square$

Now we compare the negative components between a regular production $\boldsymbol{\alpha}_x^*$ and an arbitrary production $\boldsymbol{\alpha}_x$ for the same bucket x. We show the regular cancellation cost of the former is no greater than that of the latter.

Lemma 3.40 *For any $0 \leq x \leq d-1$, let $\boldsymbol{q}_x$ be the vector such that $\boldsymbol{q}_x[x] = \ell$ and $\boldsymbol{q}_x[j] = 0$ for all $j \neq x$. Let $\boldsymbol{u}_x = \boldsymbol{\alpha}_x^* - \boldsymbol{q}_x$ and $\boldsymbol{v}_x = \boldsymbol{\alpha}_x - \boldsymbol{q}_x$. Then $rcost(\boldsymbol{u}_x) \leq rcost(\boldsymbol{v}_x)$.*

Proof. For any $I_x = (i_0, \cdots, i_k) \in \mathcal{I}_x$, define J_x as the prefix sum of I_x, i.e., $J_x[i] = \sum_{j=0}^{i} I_x[i]$ for all $0 \leq i \leq k$. Let $\boldsymbol{p}_i$ be the vector such that $\boldsymbol{p}_i[i] = -1$ and $\boldsymbol{p}_i[j] = 0$ for all $j \neq i$. Let $q_x = \lfloor x/\ell \rfloor$ and $r_x = x \bmod \ell$. Let y be an non-negative integer to represent indices of components. We prove this lemma by induction on y. The base case is when $y \leq r_x$. In this case

$$\begin{aligned}
& \boldsymbol{u}_y = (-y, 0, \cdots, 0) \\
& \boldsymbol{v}_y = (-i_0, \underbrace{0, \cdots, 0}_{i_0 - 1}, \cdots, -i_k, \underbrace{0, \cdots, 0}_{i_k - 1}, 0, \cdots, 0),
\end{aligned}$$

where $\sum_{j=0}^{k} i_j = y$ and all non-zero components of $\boldsymbol{v}_y$ have indices no smaller than 0. By Corollary 3.35 and Lemma 3.39,

$$\begin{aligned} rcost(\boldsymbol{u}_y) &= y \\ rcost(\boldsymbol{v}_y) &= i_0 + \sum_{j=1}^{k} i_j \cdot rcost(\boldsymbol{p}_{J_y[j-1]}) \\ &\geq \sum_{j=0}^{k} i_j \cdot rcost(\boldsymbol{p}_0) = y \end{aligned}$$

Thus, $rcost(\boldsymbol{u}_y) \leq rcost(\boldsymbol{v}_y)$ when $y \leq r_x$.

Assume $rcost(\boldsymbol{u}_y) \leq rcost(\boldsymbol{v}_y)$ for all $y < x$. Now consider the inductive step when $y = x$.

$$\begin{aligned} \boldsymbol{u}_x &= (-r_x, \underbrace{0, \cdots, 0}_{r-1}, \cdots, -\ell, \underbrace{0, \cdots, 0}_{\ell-1}, 0, \cdots, 0) \\ \boldsymbol{v}_x &= (-i_0, \underbrace{0, \cdots, 0}_{i_0-1}, \cdots, -i_k, \underbrace{0, \cdots, 0}_{i_k-1}, 0, \cdots, 0), \end{aligned}$$

where $\sum_{j=0}^{k} i_j = x$. Let $\boldsymbol{u}'_x = \ell \boldsymbol{p}_{x-\ell}$. Then

$$\boldsymbol{u}_x = \boldsymbol{u}_{x-\ell} + \boldsymbol{u}'_x$$

Let w be the index such that $J_x[w-1] < x-\ell$ and $J_x[w] \geq x-\ell$. Let $\delta = J_x[w]-(x-\ell)$. Note

$$\delta = J_x[w] - (x-\ell) < J_x[w] - J_x[w-1] = i_w \leq \ell$$

Let $I' = (i_0, i_1, \cdots, i_{w-1}, i_w - \delta)$. Then $I' \in \mathcal{I}_{x-\ell}$ because $1 \leq i_j \leq \ell$ for any $0 \leq j < w$, $1 \leq i_w - \delta \leq \ell$, and $\sum_{j=0}^{w} i_j - \delta = x - \ell$. Let $\boldsymbol{v}_{x-\ell} = \boldsymbol{\alpha}_{x-\ell,I'} - \boldsymbol{q}_{x-\ell}$, $\boldsymbol{v}'_x = \delta \boldsymbol{p}_w$, and

$\boldsymbol{v}''_x = \boldsymbol{v}_x - \boldsymbol{v}_{x-\ell} - \boldsymbol{v}'_x$. Then

$$\sum_{j=0}^{d-1} \boldsymbol{u}'_x[j] = -\ell$$
$$\sum_{j=0}^{d-1} \boldsymbol{v}'_x[j] = -\delta$$
$$\sum_{j=0}^{d-1} \boldsymbol{v}''_x[j] = -(\ell - \delta)$$

Note $\boldsymbol{u}_{x-\ell}, \boldsymbol{u}', \boldsymbol{v}_{x-\ell}, \boldsymbol{v}'_x, \boldsymbol{v}''_x \leq \boldsymbol{0}$. By Corollary 3.35,

$$rcost(\boldsymbol{u}_x) = rcost(\boldsymbol{u}_{x-\ell}) + rcost(\boldsymbol{u}'_x) \tag{3.19}$$
$$rcost(\boldsymbol{v}_x) = rcost(\boldsymbol{v}_{x-\ell}) + rcost(\boldsymbol{v}'_x) + rcost(\boldsymbol{v}''_x) \tag{3.20}$$

By induction hypothesis,

$$rcost(\boldsymbol{u}_{x-\ell}) \leq rcost(\boldsymbol{v}_{x-\ell}) \tag{3.21}$$

By Lemma 3.39,

$$rcost(\boldsymbol{u}'_x) = \ell \cdot rcost(\boldsymbol{p}_{x-\ell}) = \ell \cdot 2^{q_x - 1}\left(1 + \frac{r_x}{\ell}\right) \tag{3.22}$$

We finish the comparison by case analysis depending on the value of δ. If $\delta = 0$, then $\boldsymbol{v}'_x = \boldsymbol{0}$ and all the non-zero components in $\boldsymbol{v}''_x$ have indices no smaller than $x - \ell$ because

I_x is ℓ-bounded. By Corollary 3.35 and Lemma 3.39,

$$\begin{aligned}
& rcost(\boldsymbol{v}'_x) + rcost(\boldsymbol{v}''_x) && (3.23)\\
= \;& 0 + \sum_{j=w+1}^{k} i_j \cdot rcost(\boldsymbol{p}_{J[j-1]}) \\
\geq \;& \sum_{j=w+1}^{k} i_j \cdot rcost(\boldsymbol{p}_{x-\ell}) \\
= \;& \ell \cdot rcost(\boldsymbol{p}_{x-\ell}) && (3.24)
\end{aligned}$$

Combining (3.22) and (3.24) gives $rcost(\boldsymbol{u}'_x) \leq rcost(\boldsymbol{v}'_x + \boldsymbol{v}''_x)$ when $\delta = 0$.

If $\delta > 0$, then $\boldsymbol{v}'_x \neq \boldsymbol{0}$. Because I_x is ℓ-bounded, all the non-zero components in $\boldsymbol{v}''_x$ have indices no smaller than $x - (\ell - \delta)$, and all the non-zero components in $\boldsymbol{v}'_x$ have indices more smaller than $x - (2\ell - \delta)$. By Corollary 3.35 and Lemma 3.39,

$$\begin{aligned}
& rcost(\boldsymbol{v}'_x) + rcost(\boldsymbol{v}''_x) && (3.25)\\
= \;& \delta \cdot rcost(\boldsymbol{p}_{J[w-1]}) + \sum_{j=w+1}^{k} i_j \cdot rcost(\boldsymbol{p}_{J[j-1]}) \\
\geq \;& \delta \cdot rcost(\boldsymbol{p}_{x-2\ell+\delta}) + (\ell - \delta) \cdot rcost(\boldsymbol{p}_{x-\ell+\delta}) && (3.26)
\end{aligned}$$

When $0 < \delta < \ell - r_x$, applying Lemma 3.39 to (3.26) gives

$$\begin{aligned}
& rcost(\boldsymbol{v}'_x) + rcost(\boldsymbol{v}''_x) \\
\geq \;& \delta \cdot 2^{q_x - 2}\left(1 + \frac{r_x + \delta}{\ell}\right) \\
& + (\ell - \delta) \cdot 2^{q_x - 1}\left(1 + \frac{r_x + \delta}{\ell}\right) \\
= \;& (2\ell - \delta) \cdot 2^{q_x - 2}\left(1 + \frac{r_x + \delta}{\ell}\right)
\end{aligned}$$

Combining it with (3.22) gives

$$\begin{aligned}
& rcost(\boldsymbol{v}'_x) + rcost(\boldsymbol{v}''_x) - rcost(\boldsymbol{u}'_x) \\
\geq\ & 2^{q_x-2}\left((2\ell-\delta)\left(1+\frac{r_x+\delta}{\ell}\right) - 2\ell\left(1+\frac{r_x}{\ell}\right)\right) \\
=\ & 2^{q_x-2}\left(\frac{\delta(\ell-r_x-\delta)}{\ell}\right) > 0
\end{aligned}$$

When $\ell - r_x \leq \delta < \ell$, applying Lemma 3.39 to (3.26) gives

$$\begin{aligned}
& rcost(\boldsymbol{v}'_x) + rcost(\boldsymbol{v}''_x) \\
\geq\ & \delta \cdot 2^{q_x-1}\left(1+\frac{r_x+\delta-\ell}{\ell}\right) \\
& +(\ell-\delta)\cdot 2^{q_x}\left(1+\frac{r_x+\delta-\ell}{\ell}\right) \\
=\ & (2\ell-\delta)\cdot 2^{q_x-1}\left(\frac{r_x+\delta}{\ell}\right)
\end{aligned}$$

Combining it with (3.22) gives

$$\begin{aligned}
& rcost(\boldsymbol{v}'_x) + rcost)\boldsymbol{v}''_x - rcost(\boldsymbol{u}'_x) \\
\geq\ & 2^{q_x-1}\left((2\ell-\delta)\left(\frac{r_x+\delta}{\ell}\right) - d\left(1+\frac{r_x}{\ell}\right)\right) \\
=\ & 2^{q_x-1}\left(\frac{(\ell r_x - \delta r) + (2\ell\delta - \delta^2 - \ell^2)}{\ell}\right) \\
=\ & 2^{q_x-1}\left(\frac{(\ell-\delta)(r_x+\delta-\ell)}{\ell}\right) \geq 0
\end{aligned}$$

Therefore, for all possible values of δ, we have shown that

$$rcost(\boldsymbol{u}'_x) \leq rcost(\boldsymbol{v}'_x) + rcost(\boldsymbol{v}''_x) \tag{3.27}$$

Combining (3.19), (3.20), (3.21), and (3.27), we conclude that

$$rcost(\boldsymbol{u}_x) \leq rcost(\boldsymbol{v}_x)$$

$\square$

Next we show that for any non-positive vector, its cancellation cost equals its regular cancellation cost. As a consequence, an optimal activity that cancels such a vector can be constructed using only vectors in the regular vector set.

Lemma 3.41 *For any vector $\boldsymbol{p} \leq \boldsymbol{0}$, $ccost(\boldsymbol{p}) = rcost(\boldsymbol{p})$.*

Proof. We prove the lemma by showing the following statement holds: for any index x that $\boldsymbol{p}[j] = 0$ for all $j > x$, and any $r \geq 0$ such that $\boldsymbol{p} + r\boldsymbol{\alpha}_x^* \leq \boldsymbol{0}$ and $\boldsymbol{p} + r\boldsymbol{\alpha}_x \leq \boldsymbol{0}$, it is true that $ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_x^*) \leq ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_x)$ and $ccost(\boldsymbol{p}) = rcost(\boldsymbol{p})$.

Let y be a non-negative integer to indicate indices of components. We finish the proof by induction on y. The base case is when $y = 0$. Then $\mathcal{I}(0) = \{I_0^*\}$. It follows trivially that for any r such that $\boldsymbol{p} + r\boldsymbol{\alpha}_0^* \leq \boldsymbol{0}$ and $\boldsymbol{p} + r\boldsymbol{\alpha}_0 \leq \boldsymbol{0}$, we have $ccost(\boldsymbol{p}) = rcost(\boldsymbol{p})$ and $ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_0^*) = ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_0)$.

Assume the claim holds for $y < x$. Now consider the inductive step when $y = x$. The claim is trivially true when $\boldsymbol{p}[x] = 0$ because this case is reduced to the induction hypothesis. Consider the case when $\boldsymbol{p}[x] < 0$. Let $\boldsymbol{q}_x$ be the vector such that $\boldsymbol{q}_x[x] = \ell$ and $\boldsymbol{q}_x[i] = 0$ for all $i \neq x$. Let $\boldsymbol{p}' = \boldsymbol{p} + r\boldsymbol{q}_x$, $\boldsymbol{u}_x = \boldsymbol{\alpha}_x^* - \boldsymbol{q}_x$, and $\boldsymbol{v}_x = \boldsymbol{\alpha}_x - \boldsymbol{q}_x$. Then

$$\boldsymbol{p} + r\boldsymbol{\alpha}_x^* = \boldsymbol{p}' + r\boldsymbol{u}_x$$

$$\boldsymbol{p} + r\boldsymbol{\alpha}_x = \boldsymbol{p}' + r\boldsymbol{v}_x$$

Note $\boldsymbol{p}' \leq \boldsymbol{0}$ because $\boldsymbol{p} \leq \boldsymbol{0}$ and $\boldsymbol{p} + r\boldsymbol{\alpha}_x \leq \boldsymbol{0}$. Also because $\boldsymbol{u}_x \leq \boldsymbol{0}$ and $\boldsymbol{v}_x \leq \boldsymbol{0}$, by the linearity property from Corollary 3.35,

$$ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_x^*) = ccost(\boldsymbol{p}') + r \cdot ccost(\boldsymbol{u}_x) \tag{3.28}$$

$$ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_x) = ccost(\boldsymbol{p}') + r \cdot ccost(\boldsymbol{v}_x) \tag{3.29}$$

Because $\boldsymbol{u}_x \leq \boldsymbol{0}$ and $\boldsymbol{v}_x \leq \boldsymbol{0}$, and $\forall j > x-1$, $\boldsymbol{u}_x[j] = \boldsymbol{v}_x[j] = 0$, by induction hypothesis,

$$ccost(\boldsymbol{u}_x) = rcost(\boldsymbol{u}_x)$$

$$ccost(\boldsymbol{v}_x) = rcost(\boldsymbol{v}_x)$$

By Lemma 3.40, $rcost(\boldsymbol{u}_x) \leq rcost(\boldsymbol{v}_x)$, and thus

$$ccost(\boldsymbol{u}_x) \leq ccost(\boldsymbol{v}_x) \tag{3.30}$$

Combining (3.28), (3.29), and (3.30) gives

$$ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_x^*) \leq ccost(\boldsymbol{p} + r\boldsymbol{\alpha}_x)$$

It remains to show that $ccost(\boldsymbol{p}) = rcost(\boldsymbol{p})$. By Lemma 3.34, there are normal and optimal activities M^* and M to cancel $\boldsymbol{p}$ in ξ^* and ξ, respectively. Because both activities are normal, they are simple and thus only have productions for bucket x. It follows that $M_x^* = (\boldsymbol{\alpha}_x^*, -\boldsymbol{p}[x]/\ell)$, $M_x = ((\boldsymbol{\alpha}_x^1, r_1), \cdots, (\boldsymbol{\alpha}_x^k, r_k))$ where $\boldsymbol{\alpha}_x^1, \cdots, \boldsymbol{\alpha}_x^k$ are different productions for bucket x and $\sum_{j=1}^k r_j = -\boldsymbol{p}[x]/\ell$. Then $C(M_x) = C(M_x^*) = -\boldsymbol{p}[x]$, and

$$rcost(\boldsymbol{p}) = -\boldsymbol{p}[x] + rcost(\boldsymbol{p} + M_x^*) \tag{3.31}$$

$$ccost(\boldsymbol{p}) = -\boldsymbol{p}[x] + ccost(\boldsymbol{p} + M_x) \tag{3.32}$$

For each $1 \leq j \leq k$, let

$$\boldsymbol{q}_j = r_j \left(\frac{-\ell}{\boldsymbol{p}[x]} \boldsymbol{p} + \boldsymbol{\alpha}_x^* \right)$$

It follows that $\boldsymbol{q}_j \leq \boldsymbol{0}$ and thus

$$ccost(\boldsymbol{q}_j) \leq ccost\left(r_j \left(\frac{-\ell}{\boldsymbol{p}[x]} \boldsymbol{p} + \boldsymbol{\alpha}_x^j \right) \right)$$

Summing over j and applying the linearity property from Corollary 3.35 gives

$$\begin{aligned} & ccost(\boldsymbol{p} + M_x^*) && (3.33) \\ = \; & ccost\left(\sum_{j=1}^{k} \boldsymbol{q}_j \right) \\ = \; & \sum_{j=1}^{k} ccost(\boldsymbol{q}_j) \\ \leq \; & \sum_{j=1}^{k} ccost\left(r_j \left(\frac{-\ell}{\boldsymbol{p}[x]} \boldsymbol{p} + \boldsymbol{\alpha}_x^j \right) \right) \\ = \; & ccost(\boldsymbol{p} + M_x) && (3.34) \end{aligned}$$

Because $\boldsymbol{p} + M_x^* \leq \boldsymbol{0}$ and $(\boldsymbol{p} + M_x^*)[j] = 0$ for all $j > x - 1$, by induction hypothesis,

$$rcost(\boldsymbol{p} + M_x^*) = ccost(\boldsymbol{p} + M_x^*) \tag{3.35}$$

Combining (3.31), (3.32), (3.34), and (3.35) gives

$$rcost(\boldsymbol{p}) \leq ccost(\boldsymbol{p})$$

We know $rcost(\boldsymbol{p}) \geq ccost(\boldsymbol{p})$ because $U_\ell^* \subset U_\ell$. Therefore, we conclude that

$$rcost(\boldsymbol{p}) = ccost(\boldsymbol{p})$$

$\square$

Now we show that regular cancellation cost of the regular production vector is no greater than that of any production vector for the same bucket. In addition, for the regular production vector, its cancellation cost equals its regular cancellation cost. This implies that choosing the regular production as the intermediate stop and cancelling it with vectors from the regular vector set gives an optimal activity.

Lemma 3.42 *Let $d, \ell \in \mathbb{Z}^+$ be the dimension and width, respectively. Then $ccost(\boldsymbol{\alpha}^*_{d-1}) \leq ccost(\boldsymbol{\alpha}_{d-1})$, $rcost(\boldsymbol{\alpha}^*_{d-1}) \leq rcost(\boldsymbol{\alpha}_{d-1})$, and in particular, $ccost(\boldsymbol{\alpha}^*_{d-1}) = ccost(\boldsymbol{\alpha}_{d-1}) = rcost(\boldsymbol{\alpha}^*_{d-1}) = rcost(\boldsymbol{\alpha}_{d-1}) = d-1-\ell$ if $d-1 \leq \ell$.*

Proof. We prove the lemma for bounded TPS here, and the same argument directly applies to regular TPS. By Lemma 3.34, there exist normal and optimal activities M^* and M that cancel $\boldsymbol{\alpha}^*_x$ and $\boldsymbol{\alpha}_x$, respectively. Thus, $C(M^*) = ccost(\boldsymbol{\alpha}^*_{d-1})$ and $C(M) = ccost(\boldsymbol{\alpha}_{d-1})$. When $d-1 \leq \ell$, it is not hard to see that $C(M^*) = C(M) = d-1-\ell$, because the sum of positive components ℓ is no smaller than the sum of negative components x in both vectors, and thus only $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ vectors are included in M^* and M. Thus, $ccost(\boldsymbol{\alpha}^*_{d-1}) = ccost(\boldsymbol{\alpha}_{d-1}) = d-1-\ell$ in this case.

Now consider the case when $d-1 > \ell$. Because M^* and M are normal and $\boldsymbol{\alpha}^*_{d-1}[d-1] = \boldsymbol{\alpha}_{d-1}[d-1] = \ell$, both activities must first include some $\boldsymbol{\beta}$ vectors such that the ℓ units of supply in bucket $d-1$ are transferred to lower buckets and cancel ℓ units of demand there. Call these sub-activities M^*_{pre} and M_{pre}, and the resulting vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, respectively. Because $C(M^*_{pre}) = C(M_{pre}) = 0$, we have

$$C(M^*) = ccost(\boldsymbol{u}) \tag{3.36}$$

$$C(M) = ccost(\boldsymbol{v}) \tag{3.37}$$

Let $\boldsymbol{q}_x$ be the vector such that $\boldsymbol{q}_x[x] = \ell$ and $\boldsymbol{q}_x[j] = 0$ for all $j \neq x$. Then $\boldsymbol{u} = \boldsymbol{\alpha}^*_{d-1-\ell} - \boldsymbol{q}_{d-1-\ell}$ and $\boldsymbol{v} = \boldsymbol{\alpha}_{d-1-\ell} - \boldsymbol{q}_{d-1-\ell}$, and thus by Lemma 3.40, we have

$$rcost(\boldsymbol{u}) \leq rcost(\boldsymbol{v}) \tag{3.38}$$

Also because $\boldsymbol{u}, \boldsymbol{v} \leq \boldsymbol{0}$, by Lemma 3.41, we have

$$ccost(\boldsymbol{u}) = rcost(\boldsymbol{u}) \tag{3.39}$$

$$ccost(\boldsymbol{v}) = rcost(\boldsymbol{v}) \tag{3.40}$$

Combining (3.36) to (3.40) gives

$$C(M^*) \leq C(M),$$

and thus

$$ccost(\boldsymbol{\alpha}^*_{d-1}) \leq ccost(\boldsymbol{\alpha}_{d-1})$$

□

Lemma 3.43 *Let $d, \ell \in \mathbb{Z}^+$ be the dimension and width, respectively. Then $ccost(\boldsymbol{\alpha}^*_{d-1}) = rcost(\boldsymbol{\alpha}^*_{d-1})$, which is $d-1-\ell$ if $d-1 \leq \ell$, and is $2^{(\lfloor (d-1)/\ell \rfloor - 1)}(\ell + (d-1) \bmod \ell) - \ell$ if $d-1 > \ell$.*

Proof. By Lemma 3.42, $ccost(\boldsymbol{\alpha}^*_{\boldsymbol{d-1}}) = rcost(\boldsymbol{\alpha}^*_{\boldsymbol{d-1}}) = d-1-\ell$ when $d-1 \leq \ell$. Now consider the case when $d-1 > \ell$. By Lemma 3.34, there exists normal and optimal activity M^* that cancels $\boldsymbol{\alpha}^*_{d-1}$. Thus $C(M^*) = \omega(-\boldsymbol{\alpha}^*_{d-1})$. Because M^* is normal and $\boldsymbol{\alpha}^*_{d-1}[d-1] = \ell$, M^* must first include some $\boldsymbol{\beta}$ vectors such that the ℓ units of supply in bucket $d-1$ are transferred to bucket $d-1-\ell$ and cancel ℓ units of demand there. Such

$\boldsymbol{\beta}$ vectors have 0 cost and the resulting vector $\boldsymbol{u} \leq \boldsymbol{0}$. Therefore, by Lemma 3.41,

$$C(M^*) = ccost(\boldsymbol{u}) = rcost(\boldsymbol{u}) \tag{3.41}$$

Note $\boldsymbol{u} \leq \boldsymbol{0}$. There are $(d-1) \bmod \ell$ negative tokens in bucket 0 and ℓ negative tokens in each of the buckets $d-1-i\ell$ for $2 \leq i \leq \lfloor (d-1)/\ell \rfloor$. Therefore by Corollary 3.35 and Lemma 3.39,

$$\begin{aligned} rcost(\boldsymbol{u}) &= (d-1) \bmod \ell \\ &\quad +\ell \cdot \sum_{i=2}^{\lfloor (d-1)/\ell \rfloor} 2^{\lfloor (d-1)/\ell \rfloor - i} \left(1 + \frac{(d-1) \bmod \ell}{\ell}\right) \\ &= 2^{(\lfloor (d-1)/\ell \rfloor - 1)}(\ell + (d-1) \bmod \ell) \end{aligned}$$

Therefore when $d-1 > \ell$,

$$C(M^*) = 2^{(\lfloor (d-1)/\ell \rfloor - 1)}(\ell + (d-1) \bmod \ell) \tag{3.42}$$

Combining (3.41) and (3.42) gives when $d-1 > \ell$,

$$ccost(\boldsymbol{u}) = rcost(\boldsymbol{u}) = 2^{(\lfloor (d-1)/\ell \rfloor - 1)}(\ell + (d-1) \bmod \ell)$$

□

Finally, putting together Lemmas 3.38, 3.42, and 3.43, we derive the closed form formula for the minimum production cost.

Theorem 3.44 (Minimum Production Cost) *Let $d, \ell \in \mathbb{Z}^+$ be the dimension and width, respectively. Then a normal activity for the minimum token production cost problem in the bounded TPS can be constructed by using only vectors in U_ℓ^* and $|U_\ell^*| = 3d-1$. The* *minimum token production cost is* $d-1$ *if* $d-1 \leq \ell$*, and is* $2^{(\lfloor (d-1)/\ell \rfloor -1)}(\ell + (d-1) \bmod \ell)$ *if* $d-1 > \ell$*.*

Proof. We construct an optimal token production activity with vectors in U_ℓ^*. Let $M_1 = \boldsymbol{\alpha}_{d-1}^*$ and M_2 be an activity in ξ^* such that $\boldsymbol{\alpha}_{d-1}^* \xrightarrow{M_2} \mathbf{0}$ and $C(M_2) = rcost(\boldsymbol{\alpha}_{d-1}^*)$. It is straightforward that $M_1 \| M_2 \in \mathcal{M}(d, \ell)$. By Lemma 3.38, in both ξ and ξ^*,

$$\omega(\boldsymbol{\alpha}_{d-1}) = \ell$$

Therefore,

$$C(M_1) = \ell = \omega(\boldsymbol{\alpha}_{d-1}) \tag{3.43}$$

By Lemmas 3.42 and 3.43,

$$ccost(\boldsymbol{\alpha}_{d-1}^*) \leq ccost(\boldsymbol{\alpha}_{d-1})$$
$$ccost(\boldsymbol{\alpha}_{d-1}^*) = rcost(\boldsymbol{\alpha}_{d-1}^*)$$

Therefore, in both ξ and ξ^*, we have

$$C(M_2) = rcost(\boldsymbol{\alpha}_{d-1}^*) = ccost(\boldsymbol{\alpha}_{d-1}^*) \leq ccost(\boldsymbol{\alpha}_{d-1})$$

Thus, together with the definition $ccost(\boldsymbol{\alpha}_{d-1}) = \omega(-\boldsymbol{\alpha}_{d-1})$, we have

$$C(M_2) \leq \omega(-\boldsymbol{\alpha}_{d-1}) \tag{3.44}$$

Combining (3.43) and (3.44) gives

$$C(M_1 \| M_2) = C(M_1) + C(M_2) \leq \omega(\boldsymbol{\alpha}_{d-1}) + \omega(-\boldsymbol{\alpha}_{d-1}),$$

and thus $M_1 \| M_2$ is an optimal token production activity. By Lemmas 3.42 and 3.43, its cost is

$$C(M_1\|M_2) = \begin{cases} d-1 & \text{if } d-1 \le \ell, \\ 2^{(\lfloor (d-1)/\ell \rfloor - 1)}(\ell + (d-1) \bmod \ell) & \text{otherwise.} \end{cases}$$

□

3.7.2 Lower Bound Results

Following Lemma 3.37 and Theorem 3.44, we have the following lower bound for the minimum graph weight problem.

Lemma 3.45 (Minimum graph weight) *For any $d, \ell \in \mathbb{Z}^+$ such that $d > \ell$, the minimum weight of witness graphs in $\mathcal{G}(d,\ell)$ is bounded from below as* $\min\{W(G) : G \in \mathcal{G}(d,\ell)\} \ge 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$.

Proof. By Lemma 3.37,

$$\min_{M \in \mathcal{M}(d+1,\ell)} C(M) \le \min_{G \in \mathcal{G}(d,\ell)} W(G) \tag{3.45}$$

By Theorem 3.44, for $d > \ell$,

$$\min_{M \in \mathcal{M}(d+1,\ell)} C(M) = 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell) \tag{3.46}$$

Combining (3.45) and (3.46) gives

$$\min_{G \in \mathcal{G}(d,\ell)} W(G) \ge 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$$

□

Following Lemma 3.8 and Lemma 3.45, we have the following lower bound for the minimum total load problem.

Lemma 3.46 (Minimum total load) *For any $d, \ell \in \mathbb{Z}^+$ such that $d > \ell$, the minimum total load of ORA-systems in $\mathcal{N}(d, \ell)$ is bounded from below as* $\min\{H(\sigma) : \sigma \in \mathcal{N}(d, \ell)\} \geq 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$.

Proof. By Lemma 3.8,

$$\min_{G \in \mathcal{G}(d,\ell)} W(G) \leq \min_{\sigma \in \mathcal{N}(d,\ell)} H(\sigma) \tag{3.47}$$

By Lemma 3.45, for $d > \ell$,

$$\min_{G \in \mathcal{G}(d,\ell)} W(G) \geq 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell) \tag{3.48}$$

Combining (3.47) and (3.48) gives

$$\min_{\sigma \in \mathcal{N}(d,\ell)} H(\sigma) \geq 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$$

□

3.7.3 Upper Bound on the Competitive Ratio

Finally we prove Theorem 3.3 below.

Proof of Theorem 3.3: For any ORA-system, let H be the total task load, d be the makespan of an assignment computed by GREEDY, and ℓ be the makespan of an optimal assignment. It follows that $H \geq d \geq \ell$. If $d > \ell$, by Lemma 3.46, we have

$$H \geq 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$$

If $d = \ell$, we have

$$H \geq \ell = 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$$

Therefore, in both cases, we have shown that

$$H \geq 2^{(\lfloor d/\ell \rfloor - 1)}(\ell + d \bmod \ell)$$

□

Chapter 4

Checkpointing with Unreliable Checkpoints

A list is only as strong as its weakest link.

– Donald Knuth

4.1 Introduction

4.1.1 Hardware Failures and Checkpoints

Long-running jobs may arise in a variety of computer systems and applications. Examples include but are not limited to numerically intensive computations and simulations, complicated queries and transactions in database systems, large Hadoop jobs with many sequential MapReduce rounds, and so on.

Hardware failures are to be expected in any infrastructure consisting of thousands of components. The chance of job completion without any failures becomes exponentially small with increasing job processing time. Assume an execution of a job fails. After the faulty components are fixed or replaced, the lost data is recomputed by rerunning as much of the failed job as necessary. The time spent in restart and recomputation may be substantial. Define the completion time of a job to be the elapsed time between the moment the job is first started and the moment the job is completed. The completion time of a job in a failure-free environment is called the job length, or sometimes the required processing time. It has been shown in many contexts that the expected job completion time in the presence of failures grows exponentially with the job length [25, 52].

Checkpointing is a widely-used technique in systems where failures and restarts account for a significant amount of job completion time. The basic idea is to take checkpoints of the job state from time to time. When a failure is detected, the system performs a rollback operation that restores the job state to one previously stored in a checkpoint, and recomputation begins from there. This avoids the need for an interrupted job always to be restarted from the beginning. It has been shown that checkpointing can effectively reduce the job completion time in the presence of failures. The expectation grows only linearly with the job length [25, 52].

4.1.2 Motivations on Reliability of Checkpoints

In previous work checkpoints are assumed to be stored in reliable storage and can always be accessed once they are successfully created. This assumption enables many powerful tools, such as renewal random process [25] and dynamic programming [75], to be applied for the analysis of checkpoint placements. However, the following issues motivate our study of checkpointing with unreliable checkpoints in this chapter.

The assumption of total reliability for checkpoints is never strictly true in practice. No matter how checkpoints are stored, there is some possibility that they will become corrupted and therefore not available when needed. For example, network connection failures can temporarily partition the communication network and make checkpoints stored on remote servers inaccessible. Effects of excessive heat and radiation can cause bit flips and corrupt stored checkpoints [21, 38]. The issue is whether or not the non-zero checkpoint failure probability is large enough to be significant.

Replication is one approach to control the reliability of stored checkpoints. By increasing the number of replicas, the probability of them all simultaneously failing is generally reduced, but doing so takes more processing time and thus delays the completion of the job. The only justification for incurring such cost is that the checkpoint, if needed, can save the possibly much larger cost of recomputing the data it contains. Faster job completion can sometimes be achieved by using less reliable but cheaper checkpoints.

Both replication time and checkpoint reliability are affected by many factors, such as what replication schemes are used, what the replication factor is, and how replicas are distributed throughout the clustered machines. Finding an optimal checkpointing scheme is a non-trivial task.

4.1.3 Related Work

There has been considerable amount of theoretical work in the literature on checkpoint placement. An early work by Young [76] considers regular checkpoint interval or equidistant checkpointing. Assume task failures follow the Poisson distribution with mean time between failures M, and assume the time for generating a checkpoint is g. It has been shown that when M is much larger than g, a first order approximation to the optimum checkpoint interval is $w = \sqrt{2gM}$. This form of the so-called *square root law* appears in many followup works on transaction oriented systems [14, 19, 25, 36, 62]. Gelenbe [36] generalizes this result to consider the load of the system, allowing external requests to arrive during the creation of a checkpoint or rollback recovery operation. Such requests are put into a queue and get served later. All of the work mentioned above assumes a *continuous* model where checkpoints can be placed at arbitrary points during the job execution, at a cost that is independent of their locations.

Other work [15, 75] assumes a *discrete* model where a job is viewed as a sequence of tasks, and potential checkpoints can only be created at the boundaries between consecutive tasks. The cost of creating a checkpoint can vary depending on its location. This model is sometimes referred to as the *modular program* model [62]. Toueg and Babaoglu [75] present a dynamic programming solution for deciding the optimal placement of checkpoints in the modular program model. Their algorithm runs in time $O(n^3)$ to choose from the $n - 1$ potential locations for checkpoint placement.

Nicola [62] extends several existing checkpointing models, including equidistant checkpointing, the modular program model, and a random checkpointing model where the events of checkpointing follow some random process. This study confirms again the conclusion that generally holds under various models: with sufficiently many properly placed checkpoints, the expected job completion time grows linearly in the job length, while without checkpoints the expectation grows exponentially.

A more recent study by Daly [19] revisits the problem of approximating the optimum checkpoint placement interval under the continuous model with Poisson task failures. A simplified cost function yields a first-order approximation result that is almost identical to Young's result [76], while a more complete cost function provides higher order approximations to the optimum. Another research direction is to study optimal checkpoint placements with other task failure distributions and even incomplete information about such failures [11, 35, 65].

4.1.4 Our Contributions

In this chapter, we study checkpointing with unreliable checkpoints.

In Section 4.2, we define checkpointing systems and represent job executions by a two-level Markov chain. In Section 4.3, we analyze the expected number of task runs. We show in Theorem 4.5 that during the job execution the expected number of successful runs

$\mathbf{E}[S_i]$ for task t_i satisfies the recurrence relation

$$\mathbf{E}[S_{i-1}] = \left(\frac{1-q_i}{p_i}\right)\mathbf{E}[S_i] + q_i,$$

where p_i is the success probability of task t_i and q_i is the reliability of checkpoint c_i. The base case is $\mathbf{E}[S_n] = 1$. Based on that, a closed form expression for $\mathbf{E}[S_i]$ is shown in Lemma 4.6.

In Section 4.4, we present a general framework for computing the expected job completion time. We give an explicit formula for it in Theorem 4.9. In Section 4.5, we prove an interpolation theorem that the expected completion time of the concatenation of two job segments with a checkpoint of reliability q is linear as a function of q. Because the completion times at the two extreme values $q = 0$ and $q = 1$ are easy to compute, this theorem provides a convenient way of computing the completion time for any q in the interval $[0, 1]$ since it is just the weighted sum of the two extreme values.

In Section 4.6, we define uniform checkpointing systems where all tasks are identical and have success probability p, and all checkpoints are identical and have reliability q. We show in Theorem 4.13 that the quantity $p + q$ determines the growth rate of the expected job completion time as a function of the number of tasks. The expected job completion time grows linearly when $p + q > 1$, quadratically when $p + q = 1$, and exponentially when $p + q < 1$.

In Section 4.7, we study a uniform checkpointing system where the storage for checkpoints is limited and is used to retain only the most recent many checkpoints as will fit. We show in Theorem 4.14 that when $p + q > 1$, instead of keeping all $n - 1$ checkpoints, retaining only the most recent $\Omega(\log n)$ possible checkpoints ensures that the expected job completion time remains $O(n)$.

In Section 4.8, we study checkpoint placement in the continuous model with Poisson distributed task failures, where checkpoints can be placed at arbitrary points in the job. The placement of checkpoint splits a job into tasks. Task lengths are determined by the distance between adjacent checkpoints. Represent task lengths by a length vector. We show two counter-intuitive symmetry properties. In Theorem 4.24 we prove that reversing a length vector does not change the expected job completion time. In Theorem 4.28 we prove that the expected completion time can never increase when the length vector is replaced by the average of itself and its reversal. We also conjecture that when $0 < q < 1$, an optimal checkpoint placement tends to cluster checkpoints towards the middle of the job. We prove the conjecture for the special case with two checkpoints in Theorem 4.30.

In Section 4.9, we study equidistant checkpoint placement in the continuous model. Let M be the mean time between failures and g be the cost for generating a checkpoint. We show in Theorem 4.31 that when $g \ll M$ and $p + q > 1$, a first order approximation to the optimum checkpoint placement interval is $w = \sqrt{2qgM/(2-q)}$. This result generalizes the approximation result $w = \sqrt{2gM}$ in [76] where checkpoints are totally reliable.

4.2 Problem Formalization

4.2.1 Definition of Checkpointing System

A job consists of n sequential tasks to be executed one by one on a single server. Each run of a task either succeeds or fails. The job succeeds once all tasks succeed.

Definition 4.1 (Checkpointing System) *Let $T = (t_1, t_2, \cdots, t_n)$ be an ordered set of n* tasks *in a job. Let $p : T \to (0, 1]$ be a* task success probability function *that determines the success probability for each run of a task. Let $q : T \to [0, 1]$ be a* checkpoint reliability function *that determines the success probability for each attempted access to the checkpoint placed before a task. Let $w^s : T \to \mathbb{R}^+$ be a cost function for* successful task runs, *and* $w^f : T \to \mathbb{R}^+$ *be a cost function for* failed task runs. *Let* $g : T \to \mathbb{R}^+ \cup \{0\}$ *be a cost function for* generating a checkpoint *after a successful task run, and* $\alpha : T \to \mathbb{R}^+ \cup \{0\}$ *be a cost function for* accessing a checkpoint *after a failed task run. A* checkpointing system *is a seven-tuple* $(T, p, q, w^s, w^f, g, \alpha)$.

For convenience, we use subscripts to indicate the probabilities and costs for each task, i.e., for each task $t_i \in T$, let $p_i = p(t_i)$, $q_i = q(t_i)$, $w_i^s = w^s(t_i)$, $w_i^f = w^f(t_i)$, $g_i = g(t_i)$, and $\alpha_i = \alpha(t_i)$. We also use a *length vector* $\boldsymbol{w} = (w_1^s, w_2^s, \cdots, w_n^s)$ for task lengths.

A successful job execution starts at the beginning of task t_1 and ends upon the completion of task t_n. When a run of task t_i succeeds, its output is stored into checkpoint c_{i+1}. When a run of task t_i fails, the first attempt is to access the most recent checkpoint c_i. If this attempt succeeds, the job is resumed from the beginning of t_i; otherwise, this checkpoint is assumed to be permanently damaged and an attempted access to checkpoint c_{i-1} is made. Such attempts could continuously go back to all previous checkpoints until an attempt finally succeeds. By convention, it is assumed that there is an imaginary totally reliable checkpoint before task t_1, such that restarting the job is always possible.

We may refer to a consecutive subsequence of tasks $t_i, t_{i+1}, \cdots, t_{i+k}$ together with the k checkpoints $c_{i+1}, \cdots, c_{i+k}$ in between as a *job segment*. A job segment can be regarded as a compound task. In particular, a job segment with $k = 0$ is exactly a single task. It is also obvious that concatenating two job segments with a single checkpoint results in a bigger job segment. Similarly to a single task, each run of a job segment has a positive success probability, determined by the success probability of tasks and reliability of checkpoints included.

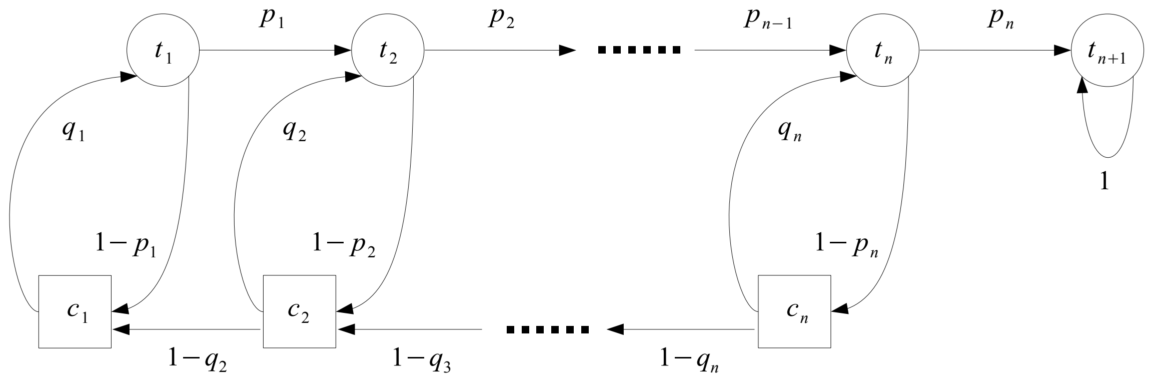

Figure 4.1: The Markov chain represents the job execution.

4.2.2 The Markov Chain Representation

We represent job execution by the Markov chain shown in Figure 4.1. It consists of $2n+1$ states for a job with n tasks. For each $1 \leq i \leq n$, state t_i represents the start of task t_i, and state c_i represents the attempted read from checkpoint c_i. For convenience, we use state t_{n+1} to represent the completion of the whole job. The weight on each directed edge represents the state transition probability.

One job execution is then represented by a random walk that starts from state t_1 and ends when it first hits state t_{n+1}. For any $1 \leq i \leq n$, let random variables S_i and F_i be the number of successful and failed runs of task t_i during the random job execution, respectively. Then S_i and F_i are the number of occurrences of the edges $t_i \to t_{i+1}$ and $t_i \to c_i$ in the corresponding random walk, respectively. Let R_i be the total number of runs for task t_i, and thus $R_i = S_i + F_i$.

Without explicitly stated otherwise, in the rest of this chapter, expectations of random variables are computed over the sample space consisting of all possible job executions.

4.2.3 Checkpoint Placements

Two commonly used checkpoint placement models in the literature, the *discrete (or modular program)* model and the *continuous* model, are special cases of the checkpointing system defined in this section.

The discrete model assumes that task lengths are per-determined. Checkpoints can only be placed at task boundaries. Task success probability might be independent of task length. A task failure is only detected at the end of this run.

The continuous model assumes that checkpoints could be placed at arbitrary points in the job. Task lengths are determined by distance between adjacent checkpoints. Task failures are assumed to follow the Poisson distribution. Task success probability depends on task length and failure rate. A task failure is detected immediately after its occurrence.

4.3 Number of Task Runs

In this section, we first show that all job executions eventually succeed via absorbing Markov chain analysis. Then we demonstrate how to compute the expected number of task runs. Knowing the number of task runs may not be sufficient to compute the job completion time, as the cost of a task run could be different depending on whether it succeeds or not. Therefore, we establish the relationship between the numbers of task runs and successful task runs. In many cases, the recurrence relation is a handy tool to analyze the job completion time. We show a recurrence relation of the expected number of successful task runs via a counting argument.

4.3.1 All Job Executions Succeed

We first show that any job execution eventually succeeds. The result is formally stated as Theorem 4.2 and the proof follows from the analysis of absorbing Markov chains. For any state in a Markov chain, it is called *absorbing* if once entered, cannot be left. A Markov chain is called *absorbing* if it has at least one absorbing state and if, from any state, it is possible to reach at least one absorbing state in finitely many steps. The following fact is known for absorbing Markov chains [24, 41].

Fact 4.1 *When an absorbing Markov chain is started in a non-absorbing state, it will eventually end up in an absorbing state.*

Based on this fact, we can easily prove the following theorem.

Theorem 4.2 *Any job execution succeeds with probability 1.*

Proof. Apparently in the Markov chain shown in Figure 4.1, there is only one absorbing state t_{n+1}. In addition, from any state, there is a simple path of length at most $2n$ to reach t_{n+1}. Therefore, this is an absorbing Markov chain. A job execution starts at state t_1. By Fact 4.1, any job execution will eventually hit the absorbing state t_{n+1} and thus succeeds. □

4.3.2 Expectation on Task Runs via Markov Chain Analysis

The expectation on the number of task runs can be computed by absorbing Markov chain analysis. We first cast the transition matrix into the *canonical form* by reordering the states so that absorbing states come before others. The canonical form is shown below.

$$\boldsymbol{P} = \left(\begin{array}{c|c} \boldsymbol{I} & 0 \\ \hline \boldsymbol{R} & \boldsymbol{Q} \end{array}\right)$$

Define the *fundamental matrix* $\boldsymbol{N} = (\boldsymbol{I} - \boldsymbol{Q})^{-1}$. The following fact is known for the fundamental matrix [24, 41].

Fact 4.3 *Each entry $N_{i,j}$ is the expected number of times that the chain is in state j before absorption when it is started in state i.*

In the job execution Markov Chain shown in Figure 4.1, there is only one absorbing state t_{n+1}. For each $1 \le i \le n$, let $\bar{p}_i = 1 - p_i$ and $\bar{q}_i = 1 - q_i$. Then the transition matrix in the canonical form is shown below.

$$
\begin{array}{c|c|ccccc|ccccc|}
 & t_{n+1} & t_1 & t_2 & \cdots & t_{n-1} & t_n & c_1 & c_2 & \cdots & c_{n-1} & c_n \\
t_{n+1} & 1 & - & - & - & - & - & - & - & - & - & - \\
\hline
t_1 & - & - & p_1 & - & - & - & \bar{p}_1 & - & - & - & - \\
t_2 & - & - & - & \ddots & - & - & - & \bar{p}_2 & - & - & - \\
\vdots & & & & & \ddots & & & & \ddots & & \\
t_{n-1} & - & - & - & - & - & p_{n-1} & - & - & - & \bar{p}_{n-1} & - \\
t_n & p_n & - & - & - & - & - & - & - & - & - & \bar{p}_n \\
\hline
c_1 & - & q_1 & - & - & - & - & - & - & - & - & - \\
c_2 & - & - & q_2 & - & - & - & \bar{q}_2 & - & - & - & - \\
\vdots & & & & \ddots & & & & \ddots & & & \\
c_{n-1} & - & - & - & - & q_{n-1} & - & - & - & \ddots & - & - \\
c_n & - & - & - & - & - & q_n & - & - & - & \bar{q}_n & -
\end{array}
$$

Let us consider a simple example with $n = 3$ tasks, where $p_1 = 0.4$, $p_2 = 0.6$, $p_3 = 0.8$, $q_1 = 1$, $q_2 = 0.7$, and $q_3 = 0.4$. Then for this example, we have

$$
\boldsymbol{Q} = \left(\begin{array}{ccc|ccc}
- & 0.4 & - & 0.6 & - & - \\
- & - & 0.6 & - & 0.4 & - \\
- & - & - & - & - & 0.2 \\
\hline
1 & - & - & - & - & - \\
- & 0.7 & - & 0.3 & - & - \\
- & - & 0.4 & - & 0.6 & -
\end{array}\right)
$$

$$
\boldsymbol{N} = (\boldsymbol{I} - \boldsymbol{Q})^{-1} = \left(\begin{array}{ccc|ccc}
3.1875 & 1.9167 & 1.25 & 2.1875 & 0.9167 & 0.25 \\
0.6875 & 1.9167 & 1.25 & 0.6875 & 0.9167 & 0.25 \\
0.1875 & 0.25 & 1.25 & 0.1875 & 0.25 & 0.25 \\
\hline
3.1875 & 1.9167 & 1.25 & 3.1875 & 0.9167 & 0.25 \\
1.4375 & 1.9167 & 1.25 & 1.4375 & 1.9167 & 0.25 \\
0.9375 & 1.25 & 1.25 & 0.9375 & 1.25 & 1.25
\end{array}\right)
$$

As a job execution starts at state t_1, the first row of $\boldsymbol{N}$ provides useful information on the expected number of task runs. In this example, we learned that $\mathbf{E}[R_1] = 3.1875$, $\mathbf{E}[R_2] = 1.9167$, $\mathbf{E}[R_3] = 1.25$.

4.3.3 Task Runs and Successful Task Runs

We establish the following fact between the expected number of task runs and the expected number of successful task runs.

Fact 4.4 *For any* $1 \le i \le n$, $\mathbf{E}[R_i] = \mathbf{E}[S_i]/p_i$ *and* $\mathbf{E}[F_i] = (1/p_i - 1)\,\mathbf{E}[S_i]$.

Proof. Because each independent run of task t_i succeeds with probability p_i, it follows from Bernoulli trials that for any $k \in \mathbb{Z}^+$,

$$\mathbf{E}[R_i \mid S_i = k] = \frac{k}{p_i}$$

Thus by the law of total expectation,

$$\begin{aligned}
\mathbf{E}[R_i] &= \mathbf{E}[\mathbf{E}[R_i \mid S_i]] \\
&= \sum_{k \in \mathbb{Z}^+} \mathbf{E}[R_i \mid S_i = k] \cdot \Pr[S_i = k] \\
&= \sum_{k \in \mathbb{Z}^+} \frac{k}{p_i} \Pr[S_i = k] \\
&= \frac{\mathbf{E}[S_i]}{p_i}
\end{aligned}$$

Therefore, by linearity of expectation,

$$\begin{aligned}
\mathbf{E}[F_i] &= \mathbf{E}[R_i - S_i] \\
&= \mathbf{E}[R_i] - \mathbf{E}[S_i] \\
&= \left(\frac{1}{p_i} - 1\right) \mathbf{E}[S_i]
\end{aligned}$$

□

4.3.4 Recurrence Relation of Expectation on Successful Task Runs

The result on the recurrence relation of $\mathbf{E}[S_i]$ is stated in Theorem 4.5. When $q_i = 1$ the expression reduces to $\mathbf{E}[S_{i-1}] = 1$, corresponding to the case with totally reliable checkpoints. When $q_i = 0$ the expression reduces to $\mathbf{E}[S_{i-1}] = \mathbf{E}[S_i]/p_i$, corresponding to the case without checkpoints.

Theorem 4.5 *Let n be the number of tasks in the job. Let random variable S_i be the number of successful runs of task t_i in a random job execution. Then $\mathbf{E}[S_n] = 1$ and for any $2 \leq i \leq n$,*

$$\mathbf{E}[S_{i-1}] = \left(\frac{1-q_i}{p_i}\right)\mathbf{E}[S_i] + q_i$$

Proof. $S_n = 1$ because the job execution succeeds once task t_n succeeds. Thus $\mathbf{E}[S_n] = 1$. Now for $2 \leq i \leq n+1$, consider task t_{i-1}. In order to finish the whole job, task t_{i-1} succeeds at least once. In addition, each time when an attempted access to checkpoint c_i fails, task t_{i-1} needs to be successfully re-executed to restore its output. Let D_i be the number of times that checkpoint c_i fails. Then, we have for all $2 \leq i \leq n+1$,

$$\mathbf{E}[S_{i-1}] = 1 + \mathbf{E}[D_i] \tag{4.1}$$

Consider checkpoint c_i. Each time when task t_i fails or an attempted access to checkpoint c_{i+1} fails, an attempted access to checkpoint c_i is made. With probability of $(1-q_i)$, such an attempt fails. Let F_i be the number of times task t_i fails. Then we have

$$\mathbf{E}[D_i] = (1-q_i)(\mathbf{E}[F_i] + \mathbf{E}[D_{i+1}]) \tag{4.2}$$

By Fact 4.4,

$$\mathbf{E}[F_i] = \left(\frac{1}{p_i} - 1\right)\mathbf{E}[S_i] \tag{4.3}$$

Combining (4.1), (4.2), and (4.3) gives for all $2 \le i \le n$,

$$\begin{aligned}
\mathbf{E}[S_{i-1}] &= 1 + \mathbf{E}[D_i] \\
&= 1 + (1 - q_i)(\mathbf{E}[F_i] + \mathbf{E}[D_{i+1}]) \\
&= 1 + (1 - q_i)\left(\left(\frac{1}{p_i} - 1\right)\mathbf{E}[S_i] + \mathbf{E}[S_i] - 1\right) \\
&= \frac{1 - q_i}{p_i}\mathbf{E}[S_i] + q_i
\end{aligned}$$

□

Applying Theorem 4.5 to the example in Section 4.3.2 gives $\mathbf{E}[S_1] = 1.275$, $\mathbf{E}[S_2] = 1.15$, and $\mathbf{E}[S_3] = 1$. Applying Fact 4.4, it follows that $\mathbf{E}[R_1] = 3.1875$, $\mathbf{E}[R_2] = 1.9167$, $\mathbf{E}[R_3] = 1.25$, in agreement with the result obtained from the Markov chain analysis.

The following closed form expression is easily derived from the recurrence relation.

Lemma 4.6 *Let n be the number of tasks in the job. Let random variable S_i be the number of successful runs of task t_i in a random job execution. Then*

$$\mathbf{E}[S_i] = \begin{cases} 1 & \textit{if } i = n, \\ (1 - q_n)/p_n + q_n & \textit{if } i = n - 1, \\ \prod_{j=i+1}^{n} \frac{1-q_j}{p_j} + \sum_{j=i+2}^{n} \left[q_j \left(\prod_{k=i+1}^{j-1} \frac{1-q_k}{p_k} \right) \right] + q_{i+1} & \textit{otherwise.} \end{cases}$$

4.4 Job Completion Time

In this section, we present a general framework to compute the expected job completion time in a checkpointing system.

4.4.1 Some Facts about Bernoulli Trials

We first present and prove several facts on Bernoulli trials. Let p be the probability of success. Let w^f and w^s be the costs of a failed trial and a successful trial, respectively. Consider the Bernoulli process where independent trials are repeated until the first success. In other words, an event is a random process such that one successful trial follows some non-negative number of failed trails. The sample space includes all such events where the number of failures are non-negative integers. Let random variable W be the cost of this Bernoulli process. We establish the relation between $\mathbf{E}[W]$, p, w^f, and w^s.

Fact 4.7 $\mathbf{E}[W] = (1/p - 1)w^f + w^s$.

Proof. Let random variable R be the total number of trials. Then it is known that for Bernoulli trials,

$$\mathbf{E}[R] = \frac{1}{p} \tag{4.4}$$

Let $k \in \mathbb{Z}^+$. Then we have

$$\mathbf{E}[W \mid R = k] = (k-1)w^f + w^s \tag{4.5}$$

By the law of total expectation, we have

$$\begin{aligned} \mathbf{E}[W] &= \mathbf{E}[\mathbf{E}[W \mid R = k]] \\ &= \sum_{k \in \mathbb{Z}^+} [\mathbf{E}[W \mid R = k] \cdot \Pr(R = k)] \end{aligned} \tag{4.6}$$

Combining (4.5) and (4.6) gives

$$\begin{aligned}
\mathbf{E}[W] &= \sum_{k\in\mathbb{Z}^+} \left[\left((k-1)w^f + w^s\right)\cdot \Pr(R=k)\right] \\
&= w^f \sum_{k\in\mathbb{Z}^+} \left[(k-1)\cdot \Pr(R=k)\right] + w^s \sum_{k\in\mathbb{Z}^+} \left[\Pr(R=k)\right] \\
&= (\mathbf{E}[R]-1)w^f + w^s \qquad (4.7)
\end{aligned}$$

Finally combining (4.4) and (4.7) gives

$$\mathbf{E}[W] = \left(\frac{1}{p} - 1\right) w^f + w^s$$

□

Let $k \in \mathbb{Z}^+$. We extend Fact 4.7 to include more than one success in an event. Consider the Bernoulli process where independent trials are repeated until the first k successes. In other words, an event is a random process such that, the pattern of one successful trial following some non-negative number of failed trails, repeats for k times. Apparently by linearity of expectation, the expected cost of such an event is $(k\,\mathbf{E}[W])$, where $\mathbf{E}[W]$ is the expected cost for the case with $k = 1$ as shown in Fact 4.7.

Let S be a random variable with sample space $\Gamma \subseteq \mathbb{Z}^+$. Let $\mathbf{E}[S]$ be its expectation over Γ. Consider the event that we repeat independent Bernoulli trials until the first k successes, where $k \in \Gamma$. Let Ω be the sample space that includes all such events for $k \in \Gamma$, where the probability measure for the events with k successes is $\Pr(S = k)$. Let random variable $W(S)$ be the cost of an event, and $\mathbf{E}[W(S)]$ be its expectation over Ω. The following fact establish the relation between $\mathbf{E}[W(S)]$, $\mathbf{E}[S]$, and $\mathbf{E}[W]$.

Fact 4.8 $\mathbf{E}[W(S)] = \mathbf{E}[S]\cdot\mathbf{E}[W]$.

Proof. Let $k \in \mathbb{Z}^+$. By linearity of expectation, we have

$$\mathbf{E}[W(S) \mid S = k] = k \cdot \mathbf{E}[W] \tag{4.8}$$

By the law of total expectation, we have

$$\begin{aligned}\mathbf{E}[W(S)] &= \mathbf{E}[\mathbf{E}[W \mid S = k]] \\ &= \sum_{k \in \Gamma} [\mathbf{E}[W(S) \mid S = k] \cdot \Pr(S = k)] \end{aligned} \tag{4.9}$$

Combining (4.8) and (4.9) gives

$$\begin{aligned}\mathbf{E}[W(S)] &= \sum_{k \in \Gamma} [k \cdot \mathbf{E}[W] \cdot \Pr(S = k)] \\ &= \mathbf{E}[S] \cdot \mathbf{E}[W]\end{aligned}$$

□

4.4.2 Expected Job Completion Time

Now we introduce costs into job executions and compute the expected job completion time. Let n be the number of tasks in the job. For each task t_i, a run of t_i is a Bernoulli trial with p_i being the probability of success. Let W_i be the cost of repeating task t_i until the first success. If we *group* runs of t_i in the job execution together, they form a sequence of Bernoulli trials that end with S_i successful runs.

Figure 4.2 illustrates the idea of grouping task runs. A successful run of a job with three tasks is shown in the top row of the figure. Each f_i and s_i represent a failed and successful run of task t_i, respectively. In each the of following three rows, we group runs of a single task together. Each group (a rectangle in Figure 4.2) consists of some non-negative number of failed runs followed by one successful run. Therefore, each group represents the Bernoulli process in which we run a single task repeatedly until the first success. The number of groups for each task t_i is its number of successful runs S_i. In the example, we have $S_1 = 4$, $S_2 = 3$ and $S_3 = 1$.

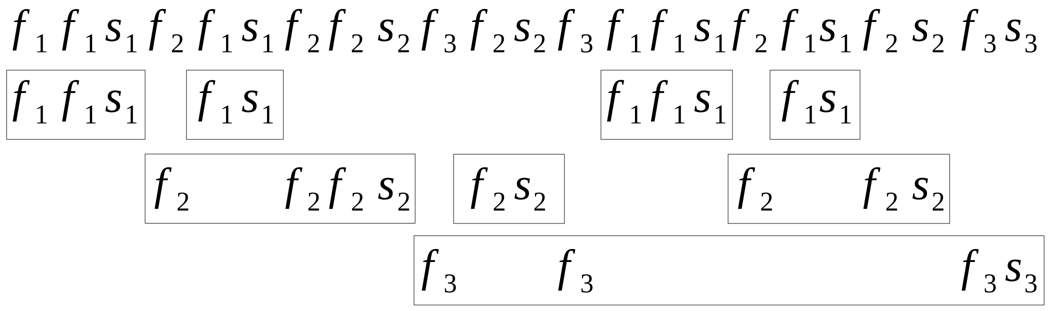
$f_1 f_1 s_1 f_2 f_1 s_1 f_2 f_2 s_2 f_3 f_2 s_2 f_3 f_1 f_1 s_1 f_2 f_1 s_1 f_2 s_2 f_3 s_3$
$f_1 f_1 s_1$ $f_1 s_1$ $f_1 f_1 s_1$ $f_1 s_1$
$f_2 \; f_2 f_2 s_2$ $f_2 s_2$ $f_2 \; f_2 s_2$
$f_3 \; f_3 \; f_3 s_3$

Figure 4.2: A successful run of a job with three tasks.

The result on the expected job completion time is summarized below.

Theorem 4.9 *Let n be the number of tasks in the job. For each task t_i, let p_i be the success probability, S_i be the number of successful runs, w_i^f be the cost of a failed run, w_i^s be the cost of a successful run, α_i be the cost of accessing a checkpoint after a failed run, and g_i be the cost of generating a checkpoint after a successful run. Then, the expected job completion time is*

$$T = \sum_{i=1}^{n} \left\{ \mathbf{E}[S_i] \cdot \left[\left(\frac{1}{p_i} - 1 \right) \left(w_i^f + \alpha_i \right) + (w_i^s + g_i) \right] \right\}$$

Proof. Because an additional cost α_i is incurred after each failed run of t_i for accessing the checkpoint, each failed run of task t_i contributes $w_i^f + \alpha_i$ to the expectation of the total cost. Similarly, because an additional cost g_i is incurred after each successful run of t_i for generating the checkpoint, each successful run of task t_i contributes $w_i^s + g_i$ to the expected total cost. Let W_i be the cost of repeatedly running task t_i until the first success.

Then by Fact 4.7, we have

$$\mathbf{E}[W_i] = \left(\frac{1}{p_i} - 1\right)\left(w_i^f + \alpha_i\right) + (w_i^s + g_i) \tag{4.10}$$

Let $W(S_i)$ be the total contribution of task t_i to the job completion time. By Fact 4.8, we have

$$\mathbf{E}[W(S_i)] = \mathbf{E}[S_i] \cdot \mathbf{E}[W_i] \tag{4.11}$$

By linearity of expectation, the expected job completion time is

$$T = \sum_{i=1}^{n} \mathbf{E}[W(S_i)] \tag{4.12}$$

Combining (4.10), (4.11), and (4.12) gives

$$T = \sum_{i=1}^{n} \left\{ \mathbf{E}[S_i] \cdot \left[\left(\frac{1}{p_i} - 1\right)\left(w_i^f + \alpha_i\right) + (w_i^s + g_i)\right]\right\}$$

□

4.5 An Interpolation Theorem

In this section, we prove an interpolation theorem on expected job completion time. In particular, the expected completion time of the concatenation of two job segments with a checkpoint of reliability q is a weighted sum of the two extreme cases where $q = 0$ and $q = 1$.

Theorem 4.10 *Let $T(q)$ be the expected completion time of the concatenation of two job segments with a checkpoint of reliability q. $T(q)$ satisfies*

$$T(q) = qT(1) + (1-q)T(0)$$

Proof. A job segment has its own success probability and expected completion time. As before, the expected completion time is defined as the expected cost for repeatedly running a job segment (Bernoulli trials with success probability p) until the first success. Such Bernoulli trials can be regarded as the execution of the job segment with an imaginary totally reliable checkpoint placed at the beginning. Then the completion time is the cost of execution that enters from the left of the job segment and leaves from the right. Let T_1, T_2 be the expected completion time of the first and second job segments, respectively. Let p_2 be the success probability of the second job segment.

Now we define another notation of cost for a job segment, called the *reverse completion time*. Again, assume there is an imaginary totally reliable checkpoint placed at the beginning of the job segment. Consider the case where a failure *after* this job segment caused fallback into it. In other words, we enter the job segment from its right end. Then the reverse completion time is defined as the cost of the execution that enters the job segment from the right and leaves from the right. Let T_1^R be the expected reverse completion time of the first job segment. Let g be the cost of generating the connecting checkpoint after each successful run of the first job segment and α be the cost of accessing the connecting checkpoint after each failed run of the second job segment. We claim that

$$T(q) = T_1 + T_2 + \left(\frac{1}{p_2} - 1\right)\alpha + (1-q)\left(\frac{1}{p_2} - 1\right)(T_1^R + g) \tag{4.13}$$

The intuition behind the formula is that both job segments need to complete, leading to the costs $T_1 + T_2$. The expected number of times that the second job segment fails is $(1/p_2 - 1)$ and thus the total access cost for the connecting checkpoint is $(1/p_2 - 1)\alpha$. The expected number of times that the connecting checkpoint fails is $(1-q)(1/p_2 - 1)$. Each time when the connecting checkpoint fails, the first job segment needs to reverse complete and generate a checkpoint. This adds the cost of $(1-q)(1/p_2 - 1)(T_1^R + g)$ to the total.

To formally prove the claim, consider the execution of the concatenation as a random walk from the left end of the first job segment and ends at the right end of the second job segment. Partition the trace of the walk by the points where it crosses the boundary of the two job segments (from either direction). See Figure 4.3 for an illustration. The vertical lines represent job segment boundaries. The cost contribution of each part of the execution is counted below.

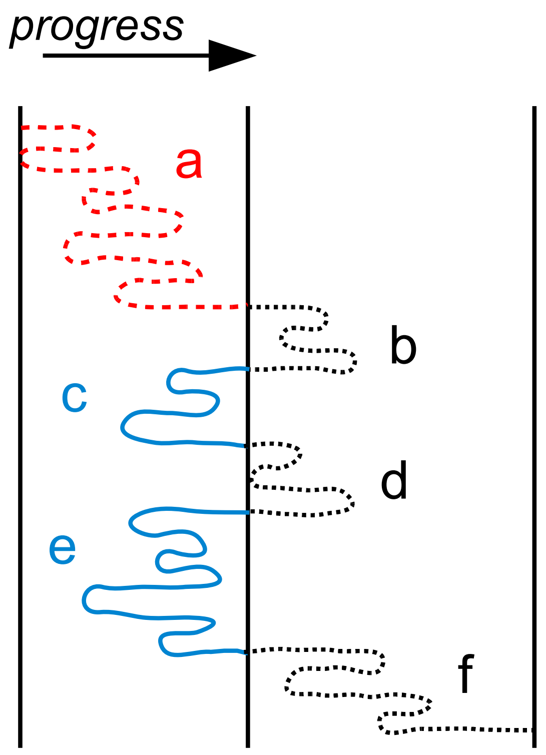

Figure 4.3: A successful run of the concatenation of two job segments.

- The cost from the beginning of the walk until it first hits the boundary, by definition, is the completion time of the first job segment. Part a in Figure 4.3 is the completion of the first job segment. The expected cost of this part is T_1.

- The part of the walk that lies to the right of the boundary, is identical to the execution that enters the second job segment from its left and leaves from the right. By definition, such cost accounts for the completion time of the second job segment. Parts b,

d and f in Figure 4.3 as a whole is the completion of the second job segment. The expected cost of this part is T_2.

- Each time when the second job segment fails, the walk hits the boundary from the right. The expected number of such failures is $(1/p_2 - 1)$. Each of these failures leads to an access cost α for the connecting checkpoint. The expected cost of this part is thus $(1/p_2 - 1)\alpha$.

- Each time when the second job segment fails, with probability $(1-q)$, the connecting checkpoint fails and thus the walk gets into the first job segment from its right. The cost from such a moment until the walk hits the boundary from the left again, by definition, is the reverse completion time of the first job segment. Following each reverse completion, the connecting checkpoint is re-generated at the cost of g. The expected number of such reverse completion is $(1-q)(1/p_2 - 1)$. Parts c and e in Figure 4.3 are two reverse completions of the first job segment. The expected cost of this part is thus $(1-q)(1/p_2 - 1)(T_1^R + g)$.

Then by linearity of expectation, (4.13) is the direct consequence from adding the above four parts of costs together. Plugging $q = 1$ and $q = 0$ into (4.13) gives

$$T(1) = T_1 + T_2 + \left(\frac{1}{p_2} - 1\right)\alpha \tag{4.14}$$

$$T(0) = T_1 + T_2 + \left(\frac{1}{p_2} - 1\right)(\alpha + T_1^R + g) \tag{4.15}$$

Finally combining (4.13), (4.14), and (4.15) gives

$$\begin{aligned} & qT(1) + (1-q)T(0) \\ = \; & T_1 + T_2 + \left(\frac{1}{p_2} - 1\right)\alpha + (1-q)\left(\frac{1}{p_2} - 1\right)(T_1^R + g) \\ = \; & T(q) \end{aligned}$$

$\square$

4.6 Uniform Checkpointing Systems

In this section, we define uniform checkpointing systems and study the asymptotic behavior of expected job completion time in such systems.

Definition 4.2 (Uniform Checkpointing System) *A* uniform *checkpointing system is a checkpointing system* $(T, p, q, w^s, w^f, g, \alpha)$ *where* $p, q, w^s, w^f, g, \alpha$ *are constant functions.*

4.6.1 Asymptotic Behavior of Expected Number of Successful Runs

Lemma 4.11 *Let* n *be the number of tasks in a uniform checkpointing system. Let* $r = (1-q)/p$*. Let random variable* S_i *be the number of successful runs of task* t_i *in a random job execution. Then for* $1 \leq i \leq n$*,*

$$\mathbf{E}[S_i] = \begin{cases} \left(1 - \frac{q}{1-r}\right) r^{n-i} + \frac{q}{1-r} & \textit{if } r \neq 1, \\ q(n-i)+1 & \textit{if } r = 1. \end{cases}$$

Proof. By Theorem 4.5, the recurrence relation is

$$\mathbf{E}[S_{i-1}] = r\,\mathbf{E}[S_i] + q \tag{4.16}$$

Substitue i with $i-1$, we obtain the recurrence

$$\mathbf{E}[S_{i-2}] = r\,\mathbf{E}[S_{i-1}] + q \tag{4.17}$$

Combining (4.16) and (4.17) gives

$$\mathbf{E}[S_{i-2}] - \mathbf{E}[S_{i-1}] = r(\mathbf{E}[S_{i-1}] - \mathbf{E}[S_i])$$

or equivalently

$$\mathbf{E}[S_{i-2}] - (r+1)\,\mathbf{E}[S_{i-1}] + r\,\mathbf{E}[S_i] = 0$$

The characteristic equation of the recurrence relation is

$$\lambda^2 - (r+1)\lambda + r = (\lambda - r)(\lambda - 1) = 0$$

The two eigenvalues of this equation are $\lambda_1 = r$ and $\lambda_2 = 1$. Therefore, we have

$$\mathbf{E}[S_i] = \begin{cases} x\lambda_1^{n+1-i} + y\lambda_2^{n+1-i} = xr^{n+1-i} + y & \text{if } r \neq 1, \\ x'\lambda_1 n + 1 - i + y'(n+1-i)\lambda_2^{n+1-i} = x' + y'(n+1-i) & \text{if } r = 1. \end{cases} \tag{4.18}$$

By Theorem 4.5, the initial values are $\mathbf{E}[S_n] = 1$ and $\mathbf{E}[S_{n-1}] = r + q$. Plugging them into (4.18) and solving for unknown coefficients gives

$$\begin{cases} x = \frac{1-r-q}{(1-r)r} \\ y = \frac{q}{1-r} \end{cases}$$

$$\begin{cases} x' = 1 - q \\ y' = q \end{cases}$$

Plugging them into (4.18) gives

$$\mathbf{E}[S_i] = \begin{cases} \left(1 - \frac{q}{1-r}\right) r^{n-i} + \frac{q}{1-r} & \text{if } r \neq 1, \\ q(n-i) + 1 & \text{if } r = 1. \end{cases}$$

□

The following theorem follows immediately from Lemma 4.11.

Theorem 4.12 *Let n be the number of tasks in a uniform checkpointing system. Let random variable S_1 be the number of successful runs of task t_1 in a random job execution. Let $r = (1-q)/p$. Then the expectation of S_1, as a function of n, is*

$$\mathbf{E}[S_1] = \begin{cases} \left(1 - \frac{q}{1-r}\right) r^{n-1} + \frac{q}{1-r} = \Theta(1) & \text{if } p+q > 1, \\ q(n-1)+1 = \Theta(n) & \text{if } p+q = 1, \\ \left(1 + \frac{q}{r-1}\right) r^{n-1} - \frac{q}{r-1} = \Theta(r^n) & \text{if } p+q < 1. \end{cases}$$

4.6.2 Asymptotic Behavior of Expected Job Completion Time

Theorem 4.13 *Let n be the number of tasks in a uniform checkpointing system. Let $r = (1-q)/p$ and $k = (1/p - 1)\left(w^f + \alpha\right) + (w^s + g)$. Then the expected job completion time, as a function of n, is*

$$T = \begin{cases} k\left[\frac{q}{1-r} n + \Theta(1)\right] = \Theta(n) & \text{if } p+q > 1, \\ k\left(\frac{qn(n-1)}{2} + n\right) = \Theta(n^2) & \text{if } p+q = 1, \\ k\left[\left(1 + \frac{q}{r-1}\right)\frac{r^n - 1}{r-1} - \frac{q}{r-1} n\right] = \Theta(r^n) & \text{if } p+q < 1. \end{cases}$$

Proof. By Theorem 4.9, we have

$$T = k \sum_{i=1}^{n} \mathbf{E}[S_i] \tag{4.19}$$

Apply Lemma 4.11 to (4.19) for the three cases with different values of $p+q$. If $p+q > 1$, then $r < 1$. Therefore,

$$\begin{aligned} T &= k\sum_{i=1}^{n}\mathbf{E}[S_i] \\ &= k\sum_{i=1}^{n}\left[\left(1-\frac{q}{1-r}\right)r^{n-i}+\frac{q}{1-r}\right] \\ &= k\left[\left(1-\frac{q}{1-r}\right)\frac{1-r^n}{1-r}+\frac{q}{1-r}n\right] \\ &= k\left[\frac{q}{1-r}n+\Theta(1)\right] = \Theta(n) \end{aligned}$$

If $p+q=1$, then $r=1$. Therefore,

$$\begin{aligned} T &= k\sum_{i=1}^{n}\mathbf{E}[S_i] \\ &= k\sum_{i=1}^{n}(q(n-i)+1) \\ &= k\left(\frac{qn(n-1)}{2}+n\right) = \Theta(n^2) \end{aligned}$$

If $p+q<1$, then $r>1$. Therefore,

$$\begin{aligned} T &= k\sum_{i=1}^{n}\mathbf{E}[S_i] \\ &= k\sum_{i=1}^{n}\left[\left(1+\frac{q}{r-1}\right)r^{n-i}-\frac{q}{r-1}\right] \\ &= k\left[\left(1+\frac{q}{r-1}\right)\frac{r^n-1}{r-1}-\frac{q}{r-1}n\right] \\ &= \Theta(r^n) \end{aligned}$$

□

Theorem 4.13 shows that the quantity $p + q$ is critical in determining the growth rate of the expected job completion time. The expected job completion time grows linearly in n when $p + q > 1$, quadratically when $p + q = 1$, and exponentially when $p + q < 1$. To avoid exponential growth in expected job completion time, system designers should keep $p + q$ no less than one.

4.7 Checkpoint Retention

Sometimes the storage for checkpoints is limited. Maintaining all $n - 1$ possible checkpoints might be undesirable in this case. Checkpoint retention polices are used to remove least useful checkpoints and make room for others. In this section, for a job with n tasks in a uniform checkpointing system, we investigate the performance of a simple policy where the most recent ℓ checkpoints are maintained and older ones are discarded. This checkpoint retention policy can be implemented by a queue and is described in Algorithm 7.

Algorithm 7: Checkpoint Retention with a Queue of capacity ℓ

```
  Input: a job of $n$ tasks, a capacity parameter $\ell$
1 define $Q$ as an empty queue of capacity $\ell$
2 while true do
3     if some task $t_i$ succeeds then
4         if $c_{i+1}$ is in $Q$ then
5             overwrite $c_{i+1}$ in $Q$      // replace corrupted checkpoint
6         else
7             if $Q.size() \geq \ell$ then
8                 $Q.dequeue()$               // remove old checkpoint
9             $Q.enqueue(c_{i+1})$            // store new checkpoint
```

We show that in an uniform checkpointing system where $p + q > 1$, by using only $\ell = \Omega(\log n)$ checkpoints, the expected job completion time grows only by a constant factor. The result is summarized in Theorem 4.14.

Theorem 4.14 *Let n be the number of tasks in a uniform checkpointing system. Let $p + q > 1$. By retaining the most recent $\ell = \Omega(\log n)$ possible checkpoints, the expected job completion time, as a function of n, is $T = \Theta(n)$.*

We first show that the probability of failing consecutive ℓ checkpoints decreases exponentially with ℓ, and thus setting $\ell = \Omega(\log n)$ drops such a probability to $1/n$. Then we show that the expected cost to complete the entire job without ever failing all maintained checkpoints grows linearly in n. Combining the two observations, we finally show that the expected cost to complete the entire job also grows linearly in n.

4.7.1 Analysis via Categorization of Random Walks

We categorize random walks and analyze the probability of falling into each category.

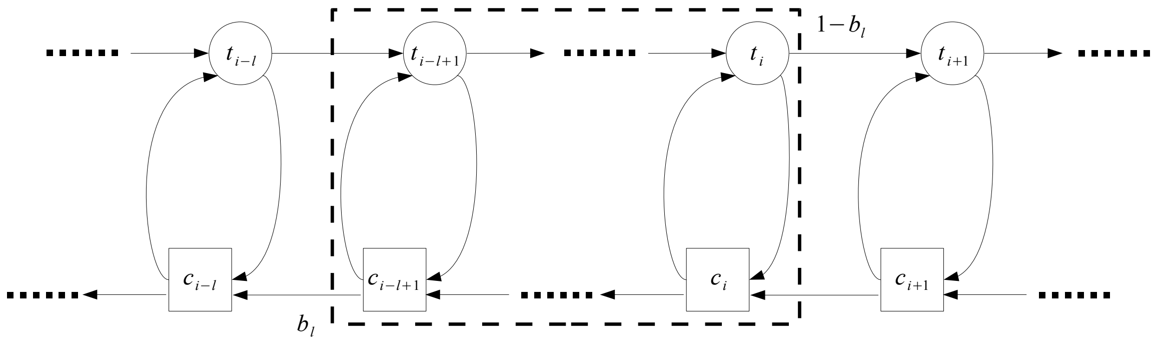

Figure 4.4: The random walk $\varphi_{i,\ell}$.

Definition 4.3 *For any $1 \le \ell < i \le n$, consider random walks that start from state c_i. Consider two states $c_{i-\ell}$ and t_{i+1}. Let $\mathcal{X}_{i,\ell}$ be the set of such random walks that first hit state $c_{i-\ell}$ and $\mathcal{Y}_{i,\ell}$ be the set of those that first hit state t_{i+1}.*

As shown in Figure 4.4, a random walk $\varphi_{i,\ell}$ that starts at c_i is categorized based on the direction it leaves the dashed rectangle. If it leaves the rectangle from the left, it belongs to $\mathcal{X}_{i,\ell}$; otherwise it belongs to $\mathcal{Y}_{i,\ell}$. The following lemma says that for any $1 \le i \le n$, a random walk $\varphi_{i,\ell}$ will eventually leave the dashed rectangle.

Lemma 4.15 *With probability 1, for any $1 \le \ell < i \le n$, random walk $\varphi_{i,\ell}$ that starts from state c_i eventually hits state $c_{i-\ell}$ or state t_{i+1}. In other words, $\Pr(\varphi_{i,\ell} \in \mathcal{X}_{i,\ell} \cup \mathcal{Y}_{i,\ell}) = 1$.*

Proof. In Figure 4.4, consider the Markov chain induced by the states in the dashed rectangle and the two ending states $c_{i-\ell}$ and t_{i+1}, both of which are regarded as sinks or absorbing states. From any non-absorbing state in the dashed rectangle, it is possible to reach at least one absorbing state in finitely many steps. Therefore, this induced Markov chain is absorbing. By Fact 4.1, when an absorbing Markov chain is started in a non-absorbing state, it will eventually end up in an absorbing state. In particular, this applies to the non-absorbing state c_i that we start the random walk $\varphi_{i,\ell}$. Thus, we have $\Pr(\varphi_{i,\ell} \in \mathcal{X}_{i,\ell} \cup \mathcal{Y}_{i,\ell}) = 1$. $\square$

As long as $i > \ell$, the starting state c_i does not affect the probability that the random walk $\varphi_{i,\ell}$ belongs to $\mathcal{X}_{i,\ell}$ or $\mathcal{Y}_{i,\ell}$. Therefore when further confusion cannot be introduced, we may ignore the subscript i from our notation.

Definition 4.4 *Let $\varphi_{i,\ell}$ be a random walk that starts from state c_i and stops when it hits either state $c_{i-\ell}$ or t_{i+1}. Let k be the number of occurrences of state c_i in the random walk, excluding the first occurrence as the starting point. Then let $\mathcal{X}_\ell^k \subset \mathcal{X}_\ell$ and $\mathcal{Y}_\ell^k \subset \mathcal{Y}_\ell$ be the sets of random walks with k occurrences of c_i. Let $\mathcal{Z}_\ell^k = \bigcup_{j \ge k}(\mathcal{X}_\ell^j \cup \mathcal{Y}_\ell^j)$ be the set of random walks with at least k occurrences of c_i.*

It follows the definition that $\mathcal{X}_\ell = \bigcup_{k\ge 0} \mathcal{X}_\ell^k$, $\mathcal{Y}_\ell = \bigcup_{k \ge 0} \mathcal{Y}_\ell^k$, and $\mathcal{Z}_\ell^0 = \mathcal{X}_\ell \cup \mathcal{Y}_\ell$.

Definition 4.5 *Let $\varphi_{i,\ell}$ be a random walk that starts from state c_i and stops when it hits either state $c_{i-\ell}$ or t_{i+1}. Consider the moment when c_i occurs in $\varphi_{i,\ell}$ for the k-th time* *(again, excluding the first occurrence as the starting point). Let $\sigma(\varphi_{i,\ell}, k)$ be the suffix of $\varphi_{i,\ell}$ that starts at the k-th occurrence of c_i. If $\varphi_{i,\ell}$ stops before the k-th occurrence of c_i, i.e., $\varphi_{i,\ell} \notin \mathcal{Z}_\ell^k$, then $\sigma(\varphi_{i,\ell}, k)$ is not defined.*

For any k, if $\sigma(\varphi_{i,\ell}, k)$ is defined, it is also a random walk that starts from state c_i.

4.7.2 Probability of Failing ℓ Checkpoints

Definition 4.6 *Let b_ℓ be the probability that a random walk φ_ℓ fails ℓ checkpoints, i.e., $\Pr(\varphi_\ell \in \mathcal{X}_\ell) = b_\ell$, $\Pr(\varphi_\ell \in \mathcal{Y}_\ell) = 1 - b_\ell$.*

We show b_ℓ decreases exponentially with ℓ. We first prove the following recurrence relation for b_ℓ.

Lemma 4.16 *For any $1 < \ell < n$, we have*

$$\frac{1}{b_\ell} = (1-p) + \frac{p}{1-q}\frac{1}{b_{\ell-1}}$$

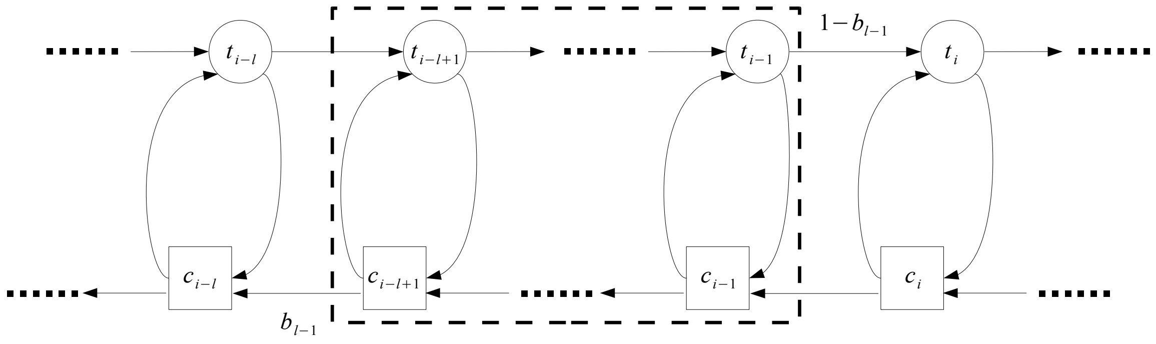

Figure 4.5: Relating φ_ℓ to $\varphi_{\ell-1}$.

Proof. By definition, a random walk φ_ℓ that starts at state c_i is in $\mathcal{X}_\ell^0$ if and only if it stops at $c_{i-\ell}$ without returning to c_i again. As shown in Figure 4.5, this happens if only if φ_ℓ directly goes to c_{i-1} via the edge $c_i \to c_{i+1}$, and then follows a random walk $\varphi_{\ell-1} \in \mathcal{X}_{\ell-1}$.

Therefore,

$$\Pr\left(\varphi_\ell \in \mathcal{X}_\ell^0\right) = (1-q)b_{\ell-1} \tag{4.20}$$

Similarly, φ_ℓ is in $\mathcal{Y}_\ell^0$ if and only if it arrives at t_i and then goes over the edge $t_i \to t_{i+1}$ to stop at t_{i+1}. There are two possible ways to go to t_i, either going directly along the edge $c_i \to t_i$, or going to c_{i-1} via the edge $c_i \to c_{i-1}$ and then following a random walk $\varphi_{\ell-1} \in \mathcal{Y}_{\ell-1}$. Therefore,

$$\begin{aligned} \Pr\left(\varphi_\ell \in \mathcal{Y}_\ell^0\right) &= [q + (1-q)(1-b_{\ell-1})]p \\ &= [1-(1-q)b_{\ell-1}]p \end{aligned} \tag{4.21}$$

By Definitions 4.4 and 4.5, it holds that for any $k \in \mathbb{N}$, φ_ℓ has exactly k occurrences of c_i if and only if φ_ℓ has at least k occurrences of c_i and the suffix $\sigma(\varphi_\ell, k)$ has no occurrence of c_i. In other words,

$$\varphi_\ell \in \mathcal{X}_\ell^k \cup \mathcal{Y}_\ell^k \iff \left(\varphi_\ell \in \mathcal{Z}_\ell^k\right) \ \wedge \ \left(\sigma(\varphi_\ell, k) \in \mathcal{X}_\ell^0 \cup \mathcal{Y}_\ell^0\right)$$

Therefore,

$$\begin{aligned} & \Pr(\varphi_\ell \in \mathcal{X}_\ell \cup \mathcal{Y}_\ell) \\ = & \sum_{k=0}^{\infty} \Pr\left(\varphi_\ell \in \mathcal{Z}_\ell^k \ \wedge \ \sigma(\varphi_\ell, k) \in \mathcal{X}_\ell^0 \cup \mathcal{Y}_\ell^0\right) \\ = & \sum_{k=0}^{\infty} \left[\Pr\left(\varphi_\ell \in \mathcal{Z}_\ell^k\right) \cdot \Pr\left(\sigma(\varphi_\ell, k) \in \mathcal{X}_\ell^0 \cup \mathcal{Y}_\ell^0\right)\right] \end{aligned} \tag{4.22}$$

Because $\sigma(\varphi_\ell, k)$ is also a random walk that starts from c_i, it follows that

$$\Pr\left(\sigma(\varphi_\ell, k) \in \mathcal{X}_\ell^0 \cup \mathcal{Y}_\ell^0\right) = \Pr\left(\varphi_\ell \in \mathcal{X}_\ell^0 \cup \mathcal{Y}_\ell^0\right) \tag{4.23}$$

Combining (4.20), (4.21), and (4.23) gives

$$\begin{aligned} & \Pr\left(\sigma(\varphi_\ell, k) \in \mathcal{X}_\ell^0 \cup \mathcal{Y}_\ell^0\right) \\ = \; & \Pr\left(\varphi_\ell \in \mathcal{X}_\ell^0\right) + \Pr\left(\varphi_\ell \in \mathcal{Y}_\ell^0\right) \\ = \; & (1-q)b_{\ell-1} + [1-(1-q)b_{\ell-1}]p \\ = \; & (1-p)(1-q)b_{\ell-1} + p \end{aligned} \tag{4.24}$$

By Lemma 4.15, φ_ℓ eventually stops and thus

$$\Pr(\varphi_\ell \in \mathcal{X}_\ell \cup \mathcal{Y}_\ell) = 1 \tag{4.25}$$

Combining (4.22), (4.24), and (4.25) gives

$$\sum_{k=0}^{\infty} \Pr\left(\varphi_\ell \in \mathcal{Z}_\ell^k\right) = \frac{1}{(1-p)(1-q)b_{\ell-1} + p} \tag{4.26}$$

Similarly, by Definitions 4.4 and 4.5, it holds that for any $k \in \mathbb{N}$, φ_ℓ has exactly k occurrences of c_i and stops at $c_{i-\ell}$ if and only if φ_ℓ has at least k occurrences of c_i, and the suffix $\sigma(\varphi_\ell, k)$ has no occurrence of c_i and stops at $c_{i-\ell}$. In other words,

$$\varphi_\ell \in \mathcal{X}_\ell^k \iff \left(\varphi_\ell \in \mathcal{Z}_\ell^k\right) \;\wedge\; \left(\sigma(\varphi_\ell, k) \in \mathcal{X}_\ell^0\right)$$

Therefore, by the same reasoning,

$$\begin{aligned} b_\ell &= \Pr(\varphi_\ell \in \mathcal{X}_\ell) \\ &= \sum_{k=0}^{\infty} \Pr\left(\varphi_\ell \in \mathcal{Z}_\ell^k \;\wedge\; \sigma(\varphi_\ell, k) \in \mathcal{X}_\ell^0\right) \\ &= (1-q)b_{\ell-1} \sum_{k=0}^{\infty} \Pr\left(\varphi_\ell \in \mathcal{Z}_\ell^k\right) \end{aligned} \tag{4.27}$$

Finally combining (4.26) and (4.27) gives

$$b_\ell = \frac{(1-q)b_{\ell-1}}{(1-p)(1-q)b_{\ell-1} + p}$$

This implies that

$$\frac{1}{b_\ell} = (1-p) + \frac{p}{1-q}\frac{1}{b_{\ell-1}}$$

□

Now we show that b_ℓ decreases exponentially with ℓ based on its recurrence relation.

Lemma 4.17 *For any* $1 < \ell < n$*, we have*

$$b_\ell \leq \left(\frac{1-q}{p}\right)^\ell \frac{p}{1-q(1-p)}$$

Proof. By Lemma 4.16, we have

$$\begin{aligned} \frac{1}{b_\ell} &= (1-p) + \frac{p}{1-q}\frac{1}{b_{\ell-1}} \\ &\geq \frac{p}{1-q}\frac{1}{b_{\ell-1}} \\ &= \left(\frac{p}{1-q}\right)^{\ell-1}\frac{1}{b_1} \end{aligned} \tag{4.28}$$

The probability that $\varphi_1 \in \mathcal{X}_1$ is

$$\begin{aligned} b_1 &= (1-q)\sum_{k=0}^{\infty}[q(1-p)]^k \\ &= \frac{1-q}{1-q(1-p)} \end{aligned} \tag{4.29}$$

Combining (4.28) and (4.29) gives

$$\begin{aligned}
b_\ell &\leq \left(\frac{1-q}{p}\right)^{\ell-1} \frac{1-q}{1-q(1-p)} \\
&= \left(\frac{1-q}{p}\right)^{\ell} \frac{p}{1-q(1-p)}
\end{aligned}$$

□

4.7.3 Job Execution with $\Omega(\log n)$ Checkpoints

We analyze job execution with $\Omega(\log n)$ checkpoints and finally prove Theorem 4.14.

Definition 4.7 *A* stage *of length ℓ consists of consecutive ℓ tasks $(t_{i-\ell+1}, \cdots, t_i)$ and the corresponding ℓ checkpoints $(c_{i-\ell+1}, \cdots, c_i)$. A run of a stage starts at the beginning of task t_i. A* stage *is* completed *if either t_i succeeds, or all the ℓ checkpoints fail. We say the stage* succeeds *in the first case and* fails *in the second case.*

A stage is equivalent to a job segment between $t_{i-\ell+1}$ and t_i plus the starting checkpoint $c_{i-\ell+1}$. The dashed rectangle shown in Figure 4.4 is a stage of length ℓ. Therefore, a run of a stage is a random walk that starts from t_i and ends at either t_{i+1} or $c_{i-\ell}$. When a stage fails, all maintained checkpoints fail and the entire job needs to start over from the very beginning.

We show that for a stage of length $\Omega(\log n)$, the expected cost when it succeeds is $O(1)$ and the failure probability is at most $1/n$. Therefore, the probability to complete n stages without ever failing back to the beginning is at least $(1 - 1/n)^n$, which is lower bounded by $1/4$ for $n \geq 2$. As a result, the expected total completion time would be $O(n)$.

Probability of Failing a Stage

Lemma 4.18 *Let d_ℓ be the probability of failing a stage of length ℓ. When $p+q>1$, we have $d_\ell \le \frac{1}{n}$ if $\ell \ge \log\left(\frac{p(1-p)}{1-q(1-p)}n\right) / \log\frac{p}{1-q}$.*

Proof. For any $\ell < i \le n$, consider random walk that starts from t_i. Let d_ℓ be the probability that it reaches $c_{i-\ell}$ before arriving at t_{i+1}. Then the event of failing a stage is equivalent for the random walk to stop at $c_{i-\ell}$. The only way to reach $c_{i-\ell}$ from t_i is to go through c_i, and thus

$$d_\ell = (1-p)b_\ell \tag{4.30}$$

By Lemma 4.17,

$$b_\ell \le \left(\frac{1-q}{p}\right)^\ell \frac{p}{1-q(1-p)} \tag{4.31}$$

Combining (4.30) and (4.31) gives

$$d_\ell \le \left(\frac{1-q}{p}\right)^\ell \frac{p(1-p)}{1-q(1-p)}$$

Letting the right hand side be no greater than $1/n$ gives

$$\left(\frac{1-q}{p}\right)^\ell \frac{p(1-p)}{1-q(1-p)} \le \frac{1}{n}$$

Note that $\frac{1-q}{p} < 1$ as $p+q>1$, and thus $\left(\frac{1-q}{p}\right)^\ell$ is monotone decreasing in ℓ. Solving for ℓ we have

$$\ell \ge \frac{\log\left(\frac{p(1-p)}{1-q(1-p)}n\right)}{\log\frac{p}{1-q}}$$

□

Corollary 4.19 *Let d_ℓ be the probability of failing a stage of length ℓ. When $p+q>1$, there exists a constant c such that $d_\ell \le \frac{1}{n}$ if $\ell \ge c\log n$.*

Proof. Let

$$c = \frac{2}{\log \frac{p}{1-q}}$$

It follows that

$$c \log n = \frac{\log (n^2)}{\log \frac{p}{1-q}} > \frac{\log \left(\frac{p(1-p)}{1-q(1-p)} n \right)}{\log \frac{p}{1-q}}$$

Therefore, by Lemma 4.18, if $\ell \geq c \log n$, we have $d_\ell \leq \frac{1}{n}$. □

Expected Cost of a Successful Stage

Lemma 4.20 *Let T^s be the expected cost for a successful stage. When $p+q > 1$, we have $T^s = O(1)$.*

Proof. A successful stage corresponds to a random walk that starts from t_i and stops at t_{i+1} without going through $c_{i-\ell}$. Now we consider the random walk that starts from $t_{i-\ell+1}$ (the first task in the stage) and stops at t_{i+1} without going through $c_{i-\ell}$. Let the expected cost of this random walk be T. We break T into two pieces: the first piece T' of starting at $t_{i-\ell+1}$ and hitting t_i, and the second piece T^s of going from t_i to t_{i+1}. By linearity of expectation,

$$T = T' + T^s \tag{4.32}$$

Given that the stage is successful, the edge $c_{i-\ell+1} \to c_{i-\ell}$ is never used, and thus it is equivalent to the case where checkpoint $c_{i-\ell+1}$ is totally reliable. By Theorem 4.13, when $p+q > 1$, the expected cost for a chain of n tasks and checkpoints is

$$T(n) = \left[\left(\frac{1}{p} - 1 \right) (w^f + \alpha) + (w^s + g) \right] \left[\frac{q}{1-r} n + \Theta(1) \right]$$

Note that $T = T(\ell + 1)$ and $T' = T(\ell)$. Therefore,

$$T^s = T - T' = T(\ell + 1) - T(\ell) = O(1) \tag{4.33}$$

□

A Time-Bounded Job Execution

To upper-bound the job completion time under the retention policy, we slightly revise the job execution with an additional timer. Give constant parameters k_1 and k_2, a timer of value $k_1 k_2 n$ is maintained. The basic idea is to start this timer and run the job from the beginning. If the execution does not succeed before the timer expires, reset the timer and restart the job from the beginning. Repeat this process until the job succeeds. We bound the expected job completion time with the additional timer and show it is in $O(n)$.

Lemma 4.21 *There exist positive constants c, k_1 and k_2 such that, with $c \log n$ most recent checkpoints and a timer of value $k_1 k_2 n$, the expected job completion time is $O(n)$.*

Proof. Consider the experiment that we start the job from the beginning until it either succeeds or fails ℓ consecutive checkpoints (i.e., fails a stage of length ℓ). We bound the probability that one experiment succeeds before the timer expires. Let I be the indicator variable such that $I = 1$ when the experiment succeeds. Let d_ℓ be the probability of failing a stage of length ℓ. Then by Corollary 4.19, there exists constant a c such that for $\ell = c \log n$,

$$d_\ell \leq \frac{1}{n}$$

Thus, we have

$$\Pr(I = 1) \geq (1 - d_\ell)^n \geq \left(1 - \frac{1}{n}\right)^n \tag{4.34}$$

It is known that $(1 - 1/n)^n$ is monotone increasing in n and approaches $1/e$ when n approaches infinity. For any job with at least one checkpoint, we have $n \geq 2$ and thus

$$\left(1 - \frac{1}{n}\right)^n \geq \left(1 - \frac{1}{2}\right)^2 = \frac{1}{4} \tag{4.35}$$

Let random variable T_1 be the cost of one experiment. T_1 consists of the cost of finishing the first $c \log n$ tasks (and thus generating the corresponding $c \log n$ checkpoints), and the cost of finishing the remaining stages. By Theorem 4.13, the expected cost to finish the first $c \log n$ tasks is $O(\log n)$. Let T^s be the expected cost for a successful stage. By Lemma 4.20, we have $T^s = O(1)$. Therefore,

$$\mathbf{E}[T_1 \mid I = 1] \leq O(\log n) + n \cdot O(1) = O(n)$$

Let k_1 be a constant that $\mathbf{E}[T_1 \mid I = 1] \leq k_1 n$ and k_2 be any positive constant that is greater than 1. By Markov's inequality, the probability that a successful experiment runs longer than $k_1 k_2 n$ is

$$\Pr(T_1 \geq k_1 k_2 n \mid I = 1) \leq \frac{\mathbf{E}[T_1 \mid I = 1]}{k_1 k_2 n} \leq \frac{1}{k_2} \tag{4.36}$$

Combining (4.34), (4.35), and (4.36) gives that the probability that an experiment succeeds before the timer expires is at least a constant,

$$\begin{aligned} \hat{p} &= \Pr(T_1 \leq k_1 k_2 n \ \wedge \ I = 1) \\ &= \Pr(T_1 \leq k_1 k_2 n \mid I = 1) \cdot \Pr(I = 1) \\ &\geq \left(1 - \frac{1}{k_2}\right) \frac{1}{4} \end{aligned}$$

Then the job execution within the $k_1 k_2 n$ timer bounds can be regarded as a process of repeated Bernoulli trials. Each trial succeeds with probability at least $\hat{p}$ and has cost at most $k_1 k_2 n$. The expected number of experiments until a success is at most $1/\hat{p} = O(1)$. Therefore, the expected job completion time is

$$T \leq \frac{k_1 k_2 n}{\hat{p}} = O(n)$$

□

Expected Job Completion Time

Finally we prove Theorem 4.14 and show the expected job completion time is $\Theta(n)$.

Proof of Theorem 4.14: The lower bound is straightforward as completing n tasks has cost at least linear in n, and thus

$$T = \Omega(n) \tag{4.37}$$

The upper bound is obtained by comparing the job execution under Algorithm 7 against the execution with the additional timer. When the timer expires, all checkpoints are discarded, the job restarts from the beginning, and all progress is lost. In contrast, normal job execution without the timer never intentionally drops progress and thus cannot have a worse expected completion time. By Lemma 4.21, with the additional timer, the expected job completion time is $O(n)$. Therefore, for normal executions without the timer, we have

$$T = O(n) \tag{4.38}$$

Finally combining (4.37) and (4.38) we conclude that

$$T = \Theta(n)$$

□

4.8 Poisson Checkpointing Systems

In this section, we define Poisson checkpointing systems and study how checkpoint placement affects the expected job completion time.

In a Poisson checkpointing system, task failures are assumed to follow the Poisson distribution. Therefore, task success probability is an exponential function of the task length w^s and failure rate λ. The cost of a failed task w^f also depends on these two parameters. A task failure is assumed to be detected immediately after it occurs. Therefore, w^f is defined to be the average length of failed runs for each task. Formally,

Definition 4.8 (Poisson Checkpointing System) *Let λ be the task failure rate. A* Poisson *checkpointing system is a checkpointing system $(T, p, q, w^s, w^f, g, \alpha)$ where for all $t_i \in T$,*

$$\begin{aligned} p_i &= e^{-\lambda w_i^s} \\ w_i^f &= \frac{1}{\lambda} - \frac{w_i^s}{e^{\lambda w_i^s} - 1} \end{aligned}$$

Therefore, a Poisson checkpointing system is defined by $(T, \lambda, q, w^s, g, \alpha)$. To simplify the notation, we may ignore the superscript in w^s and write it as w.

It follows immediately from the definition of Poisson checkpointing systems that, concatenating two tasks of lengths w_i and w_j with a checkpoint of reliability $q = 0$ results in a single task of length $w_i + w_j$. It is also true that for all $t_i \in T$ in a Poisson checkpointing system,

$$\left(\frac{1}{p_i} - 1\right) w_i^f + w_i^s = \frac{1/p_i - 1}{\lambda} = \frac{e^{\lambda w_i} - 1}{\lambda}$$

In the rest of this section, to focus on the effect of checkpoint placement on the expected job completion time, we assume q only has one value in its range and ignore the costs for generating and accessing checkpoints, i.e., $g = \alpha = 0$. When the total job length and the number of checkpoints are fixed, one wishes to minimize the expected job completion time by choosing an optimal length vector.

4.8.1 Expected Job Completion Time with Two Tasks

Let us warm up with the simplest case where a single checkpoint partitions the job into two tasks. We first show that reversing a length vector does not change the expected job completion time. Then we show that when $q \neq 0$, the optimal placement of the single checkpoint is independent of q, and it is exactly in the middle of the job.

Lemma 4.22 *Let $\boldsymbol{w} = (w_1, w_2)$ be a length vector and $\boldsymbol{w}^R = (w_2, w_1)$ be its reversal. Then for the expected job completion time, we have*

$$T(\boldsymbol{w}) = T(\boldsymbol{w}^R)$$

Proof. By Theorem 4.10,

$$\begin{aligned} T(\boldsymbol{w}) &= q[T(w_1) + T(w_2)] + (1-q)T(w_1 + w_2) \\ &= q[T(w_2) + T(w_1)] + (1-q)T(w_2 + w_1) \\ &= T(\boldsymbol{w}^R) \end{aligned}$$

□

Lemma 4.23 *Let $\boldsymbol{w} = (w_1, w_2)$ be a length vector and $\boldsymbol{w}^R = (w_2, w_1)$ be its reversal. Then for the expected job completion time, we have*

$$T(\boldsymbol{w}) \geq T\left(\frac{\boldsymbol{w} + \boldsymbol{w}^R}{2}\right)$$

The equality holds iff $q = 0$ or $w_1 = w_2$.

Proof. By Theorem 4.10,

$$\begin{aligned} T(\boldsymbol{w}) &= q[T(w_1) + T(w_2)] + (1-q)T(w_1+w_2) \\ &= q\left[\frac{e^{\lambda w_1}-1}{\lambda} + \frac{e^{\lambda w_2}-1}{\lambda}\right] + (1-q)T(w_1+w_2) \qquad (4.39) \end{aligned}$$

Similarly, we have

$$T\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2}\right) = 2q\left[\frac{e^{\lambda(w_1+w_2)/2}-1}{\lambda}\right] + (1-q)T(w_1+w_2) \qquad (4.40)$$

Combining (4.39) and (4.40) gives

$$\begin{aligned} T(\boldsymbol{w}) - T\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2}\right) &= \frac{q}{\lambda}\left[e^{\lambda w_1} + e^{\lambda w_2} - 2e^{\lambda(w_1+w_2)/2}\right] \\ &= \frac{q}{\lambda}\left[e^{\lambda w_1/2} - e^{\lambda w_2/2}\right]^2 \\ &\geq 0 \end{aligned}$$

The equality holds iff $q = 0$ or $w_1 = w_2$. Therefore, we conclude that

$$T(\boldsymbol{w}) \geq T\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2}\right)$$

If $q \neq 0$, the optimal checkpoint placement is at the middle of the job. □

4.8.2 Expected Job Completion Time with n Tasks

Now we extend the two symmetry properties to jobs with arbitrary n tasks.

Theorem 4.24 *Let $\boldsymbol{w} = (w_1, \cdots, w_n)$ be a length vector and $\boldsymbol{w}^R = (w_n, \cdots, w_1)$ be its reversal. Then for the expected job completion time, we have $T(\boldsymbol{w}) = T(\boldsymbol{w}^R)$.*

Proof. We finish the proof by induction on the number of tasks. For better notation, we add subscripts to T to explicitly indicate the number of tasks in a job. The base cases with $\ell = 0$ and $\ell = 1$ task are trivially true. For $n \geq 2$, assume the claim holds for $\ell = n - 1$ tasks, i.e., $T_{n-1}(\boldsymbol{w}) = T_{n-1}(\boldsymbol{w}^R)$ for any length vector $\boldsymbol{w}$ of length $n - 1$. Now consider the inductive step with $\ell = n$ tasks. Applying Theorem 4.10 to the checkpoint after the first task in $\boldsymbol{w}$ gives

$$\begin{aligned} T_n(w_1, \cdots, w_n) &= q\left[T_1(w_1) + T_{n-1}(w_2, \cdots, w_n)\right] \\ &\quad + (1-q)T_{n-1}(w_1 + w_2, w_3, \cdots, w_n) \end{aligned} \tag{4.41}$$

Applying Theorem 4.10 to the checkpoint before the last task in $\boldsymbol{w}^R$ gives

$$\begin{aligned} T_n(w_n, \cdots, w_1) &= q\left[T_{n-1}(w_n, \cdots, w_2) + T_1(w_1)\right] \\ &\quad + (1-q)T_{n-1}(w_n, \cdots, w_3, w_2 + w_1) \end{aligned} \tag{4.42}$$

By induction hypothesis,

$$T_{n-1}(w_2, \cdots, w_n) = T_{n-1}(w_n, \cdots, w_2) \tag{4.43}$$

$$T_{n-1}(w_1 + w_2, w_3, \cdots, w_n) = T_{n-1}(w_n, \cdots, w_3, w_2 + w_1) \tag{4.44}$$

Finally combining (4.41), (4.42), (4.43), and (4.44) gives

$$T_n(w_1, \cdots, w_n) = T_n(w_n, \cdots, w_1)$$

□

Lemma 4.25 *Let $T_n(\boldsymbol{w})$ be the expected job completion for the length vector $\boldsymbol{w}$ of length*

n. *When* $n \geq 2$, *for any length vector* $\mathbf{u}$ *and* $\mathbf{v}$ *of length* n, *we have*

$$\begin{aligned}
& T_n(\boldsymbol{u}) + T_n(\boldsymbol{v}) \\
= \; & q[T_1(u_n) + T_1(v_1)] \\
& +q[T_{n-1}(u_1, \cdots, u_{n-1}) + T_{n-1}(v_2, \cdots, v_n)] \\
& +(1-q)[T_{n-1}(u_1, \cdots, u_{n-2}, u_{n-1}+u_n) + T_{n-1}(v_1+v_2, v_3 \cdots, v_n)]
\end{aligned}$$

Proof. Applying Theorem 4.10 to the checkpoint before the last task in $(u_1, \cdots, u_n)$ gives

$$\begin{aligned}
& T_n(u_1, \cdots, u_n) \\
= \; & q\,[T_{n-1}(u_1, \cdots, u_{n-1}) + T_1(u_n)] \\
& +(1-q)T_{n-1}(u_1, \cdots, u_{n-2}, u_{n-1}+u_n)
\end{aligned} \tag{4.45}$$

Applying Theorem 4.10 to the checkpoint after the first task in $(v_1, \cdots, v_n)$ gives

$$\begin{aligned}
& T_n(v_1, \cdots, v_n) \\
= \; & q\,[T_1(v_1) + T_{n-1}(v_2, \cdots, v_n)] \\
& +(1-q)T_{n-1}(v_1+v_2, v_3, \cdots, v_n)
\end{aligned} \tag{4.46}$$

Combining (4.45) and (4.45) and rearranging terms gives

$$\begin{aligned}
& T_n(\boldsymbol{u}) + T_n(\boldsymbol{v}) \\
= \; & q[T_1(u_n) + T_1(v_1)] \\
& +q[T_{n-1}(u_1, \cdots, u_{n-1}) + T_{n-1}(v_2, \cdots, v_n)] \\
& +(1-q)[T_{n-1}(u_1, \cdots, u_{n-2}, u_{n-1}+u_n) + T_{n-1}(v_1+v_2, v_3 \cdots, v_n)]
\end{aligned}$$

□

Let $\boldsymbol{w} = (w_1, \cdots, w_n)$ be a length vector of n tasks. For the degenerate case where $n = 0$, we say $\boldsymbol{w}$ is a null vector and contains zero component. Let $\boldsymbol{w}^R = (w_n, \cdots, w_1)$ be the reversal of $\boldsymbol{w}$. For any $x, y \geq 0$, define a family of functions f_n as

$$f_n(\boldsymbol{w}, x, y) = \begin{cases} 0 & \text{if } n = 0, \\ T_n(x + w_1, w_2, \cdots, w_n) + T_n(w_1, \cdots, w_{n-1}, w_n + y) & \text{if } n \geq 1, \end{cases}$$

We show a recurrence relation and a symmetry property for f_n.

Lemma 4.26 *For $n \geq 2$, let $\boldsymbol{w} = (w_1, \cdots, w_n)$ be a length vector. Let $\boldsymbol{u} = (w_2, \cdots, w_{n-1})$ and $\boldsymbol{v} = (w_1 + w_2, w_3, \cdots, w_{n-2}, w_{n-1} + w_n)$. Then we have*

$$\begin{aligned} & f_n(\boldsymbol{w}, x, y) \\ = \; & q f_1(\mathbf{0}, w_1, w_n) + q f_1(\mathbf{0}, x + w_1, w_n + y) + q^2 f_{n-2}(\boldsymbol{u}, 0, 0) \\ & + q(1-q) f_{n-2}(\boldsymbol{u}, w_1, w_n) + q(1-q) f_{n-2}(\boldsymbol{u}, x + w_1, w_n + y) \\ & + (1-q)^2 f_{n-2}(\boldsymbol{v}, x, y) \end{aligned}$$

Proof. Let

$$a = T_{n-1}(x + w_1, w_2, \cdots, w_{n-1}) + T_{n-1}(w_2, \cdots, w_{n-1}, w_n + y) \tag{4.47}$$

$$\begin{aligned} b = \; & T_{n-1}(x + w_1, w_2, \cdots, w_{n-2}, w_{n-1} + w_n) \\ & + T_{n-1}(w_1 + w_2, w_3, \cdots, w_{n-1}, w_n + y) \end{aligned} \tag{4.48}$$

By Lemma 4.25, we have

$$\begin{aligned}
& f_n(\boldsymbol{w}, x, y) \\
= \; & T_n(x+w_1, w_2, \cdots, w_n) + T_n(w_1, \cdots, w_{n-1}, w_n + y) \\
= \; & q[T_1(w_n) + T_1(w_1)] \\
& + q[T_{n-1}(x+w_1, w_2, \cdots, w_{n-1}) + T_{n-1}(w_2, \cdots, w_{n-1}, w_n + y)] \\
& + (1-q)[T_{n-1}(x+w_1, w_2, \cdots, w_{n-2}, w_{n-1} + w_n) \\
& \quad + T_{n-1}(w_1 + w_2, w_3, \cdots, w_{n-1}, w_n + y)] \\
= \; & q f_1(\mathbf{0}, w_1, w_n) + qa + (1-q)b \qquad (4.49)
\end{aligned}$$

Applying Lemma 4.25 to (4.47) gives

$$\begin{aligned}
a \; = \; & T_{n-1}(w_2, \cdots, w_{n-1}, w_n + y) + T_{n-1}(x + w_1, w_2, \cdots, w_{n-1}) \\
= \; & q[T_1(w_n + y) + T_1(x + w_1)] \\
& + q[T_{n-2}(w_2, \cdots, w_{n-1}) + T_{n-2}(w_2, \cdots, w_{n-1})] \\
& + (1-q)[T_{n-2}(w_2, \cdots, w_{n-2}, w_{n-1} + w_n + y) \\
& \quad + T_{n-2}(x + w_1 + w_2, w_3, \cdots, w_{n-1})] \\
= \; & q f_1(\mathbf{0}, x + w_1, w_n + y) + q f_{n-2}(\boldsymbol{u}, 0, 0) \\
& + (1-q) f_{n-2}(\boldsymbol{u}, x + w_1, w_n + y) \qquad (4.50)
\end{aligned}$$

Applying Lemma 4.25 to (4.48) gives

$$\begin{aligned}
b &= T_{n-1}(w_1+w_2,w_3,\cdots,w_{n-1},w_n+y) \\
&\quad + T_{n-1}(x+w_1,w_2,\cdots,w_{n-2},w_{n-1}+w_n) \\
&= q[T_1(w_n+y)+T_1(x+w_1)] \\
&\quad +q[T_{n-2}(w_1+w_2,w_3,\cdots,w_{n-1})+T_{n-2}(w_2,\cdots,w_{n-2},w_{n-1}+w_n)] \\
&\quad +(1-q)[T_{n-2}(w_1+w_2,w_3,\cdots,w_{n-2},w_{n-1}+w_n+y) \\
&\quad\quad + T_{n-2}(x+w_1+w_2,w_3,\cdots,w_{n-2},w_{n-1}+w_n)] \\
&= qf_1(\mathbf{0},x+w_1,w_n+y)+qf_{n-2}(\boldsymbol{u},w_1,w_n) \\
&\quad +(1-q)f_{n-2}(\boldsymbol{v},x,y) \qquad (4.51)
\end{aligned}$$

Finally, combining (4.49), (4.50), and (4.51) gives

$$\begin{aligned}
& f_n(\boldsymbol{w},x,y) \\
= \ & qf_1(\mathbf{0},w_1,w_n)+qf_1(\mathbf{0},x+w_1,w_n+y)+q^2f_{n-2}(\boldsymbol{u},0,0) \\
& +q(1-q)f_{n-2}(\boldsymbol{u},w_1,w_n)+q(1-q)f_{n-2}(\boldsymbol{u},x+w_1,w_n+y) \\
& +(1-q)^2f_{n-2}(\boldsymbol{v},x,y)
\end{aligned}$$

□

Lemma 4.27 *Let $\boldsymbol{w}=(w_1,\cdots,w_n)$ be a length vector and $\boldsymbol{w}^R=(w_n,\cdots,w_1)$ be its reversal. For any $n\ge 0$ and any $x,y\ge 0$, we have*

$$f_n(\boldsymbol{w},x,y)\ge f_n\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2},\frac{x+y}{2},\frac{x+y}{2}\right)$$

Proof. We finish the proof by induction. Note that with $\ell = 0$ task, the claim is trivially true as $f_0 = 0$. When $\ell = 1$, we have $\boldsymbol{w} = (\boldsymbol{w} + \boldsymbol{w}^R)/2$. Therefore,

$$\begin{aligned} f_1(\boldsymbol{w}, x, y) &= T_1(x + w_1) + T_1(w_1 + y) = \frac{(e^{\lambda x} + e^{\lambda y})e^{\lambda w_1} - 2}{\lambda} \\ f_1\left(\frac{\boldsymbol{w} + \boldsymbol{w}^R}{2}, \frac{x+y}{2}, \frac{x+y}{2}\right) &= 2T_1\left(w_1 + \frac{x+y}{2}\right) = \frac{2e^{\lambda(x+y)/2} \cdot e^{\lambda w_1} - 2}{\lambda} \end{aligned}$$

Combining the two gives

$$\begin{aligned} & f_1(\boldsymbol{w}, x, y) - f_1\left(\frac{\boldsymbol{w} + \boldsymbol{w}^R}{2}, \frac{x+y}{2}, \frac{x+y}{2}\right) \\ = \; & \frac{e^{\lambda w_1}}{\lambda}\left(e^{\lambda x} + e^{\lambda y} - 2e^{\lambda(x+y)/2}\right) \\ = \; & \frac{e^{\lambda w_1}}{\lambda}\left(e^{\lambda x/2} - e^{\lambda y/2}\right)^2 \\ \geq \; & 0 \end{aligned}$$

Thus, we have

$$f_1(\boldsymbol{w}, x, y) \geq f_1\left(\frac{\boldsymbol{w} + \boldsymbol{w}^R}{2}, \frac{x+y}{2}, \frac{x+y}{2}\right)$$

For $n \geq 2$, assume the claim holds for jobs with $\ell < n$ tasks. Consider the inductive step with $\ell = n$ tasks. By Lemma 4.26,

$$\begin{aligned} & f_n(\boldsymbol{w}, x, y) \\ = \; & q f_1(\mathbf{0}, w_1, w_n) \\ & + q f_1(\mathbf{0}, x + w_1, w_n + y) \\ & + q^2 f_{n-2}(\boldsymbol{u}, 0, 0) \\ & + q(1-q) f_{n-2}(\boldsymbol{u}, w_1, w_n) \\ & + q(1-q) f_{n-2}(\boldsymbol{u}, x + w_1, w_n + y) \\ & + (1-q)^2 f_{n-2}(\boldsymbol{v}, x, y) \end{aligned}$$

$$\begin{aligned}
& f_n\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2}, \frac{x+y}{2}, \frac{x+y}{2}\right) \\
= \quad & q f_1\left(\mathbf{0}, \frac{w_1+w_n}{2}, \frac{w_1+w_n}{2}\right) \\
& + q f_1\left(\mathbf{0}, \frac{x+w_1+w_n+y}{2}, \frac{x+w_1+w_n+y}{2}\right) \\
& + q^2 f_{n-2}\left(\frac{\boldsymbol{u}+\boldsymbol{u}^R}{2}, 0, 0\right) \\
& + q(1-q) f_{n-2}\left(\frac{\boldsymbol{u}+\boldsymbol{u}^R}{2}, \frac{w_1+w_n}{2}, \frac{w_1+w_n}{2}\right) \\
& + q(1-q) f_{n-2}\left(\frac{\boldsymbol{u}+\boldsymbol{u}^R}{2}, \frac{x+w_1+w_n+y}{2}, \frac{x+w_1+w_n+y}{2}\right) \\
& + (1-q)^2 f_{n-2}\left(\frac{\boldsymbol{v}+\boldsymbol{v}^R}{2}, \frac{x+y}{2}, \frac{x+y}{2}\right)
\end{aligned}$$

By induction base and hypothesis, we have

$$\begin{aligned}
f_1(\mathbf{0}, w_1, w_n) & \geq f_1\left(\mathbf{0}, \frac{w_1+w_n}{2}, \frac{w_1+w_n}{2}\right) \\
f_1(\mathbf{0}, x+w_1, w_n+y) & \geq f_1\left(\mathbf{0}, \frac{x+w_1+w_n+y}{2}, \frac{x+w_1+w_n+y}{2}\right) \\
f_{n-2}(\boldsymbol{u}, 0, 0) & \geq f_{n-2}\left(\frac{\boldsymbol{u}+\boldsymbol{u}^R}{2}, 0, 0\right) \\
f_{n-2}(\boldsymbol{u}, w_1, w_n) & \geq f_{n-2}\left(\frac{\boldsymbol{u}+\boldsymbol{u}^R}{2}, \frac{w_1+w_n}{2}, \frac{w_1+w_n}{2}\right) \\
f_{n-2}(\boldsymbol{u}, x+w_1, w_n+y) & \geq f_{n-2}\left(\frac{\boldsymbol{u}+\boldsymbol{u}^R}{2}, \frac{x+w_1+w_n+y}{2}, \frac{x+w_1+w_n+y}{2}\right) \\
f_{n-2}(\boldsymbol{v}, x, y) & \geq f_{n-2}\left(\frac{\boldsymbol{v}+\boldsymbol{v}^R}{2}, \frac{x+y}{2}, \frac{x+y}{2}\right)
\end{aligned}$$

Therefore, we conclude that

$$f_n(\boldsymbol{w}, x, y) \geq f_n\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2}, \frac{x+y}{2}, \frac{x+y}{2}\right)$$

□

The following theorem is a direct consequence of Lemma 4.27.

Theorem 4.28 *Let $\boldsymbol{w}$ be a length vector and $\mathbf{w}^R$ be its reversal. Then for the expected job completion time, we have*

$$T(\boldsymbol{w}) \geq T\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2}\right)$$

Proof. It follows from Lemma 4.27 that

$$T(\boldsymbol{w}) = \frac{1}{2}f(\boldsymbol{w},0,0) \geq \frac{1}{2}f\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2},0,0\right) = T\left(\frac{\boldsymbol{w}+\boldsymbol{w}^R}{2}\right)$$

□

4.8.3 Optimal Checkpoint Placement

It has been shown that when $q = 1$, placing checkpoint in equidistant points is optimal. When q decreases, we conjecture that an optimal placement tends to cluster checkpoints towards the middle of the job.

We prove the special case of this conjecture with two checkpoints. Let (w_1, w_2, w_3) be a length vector. It follows from Theorem 4.5 that the expected number of successful runs for the three tasks are

$$\begin{aligned}
\mathbf{E}[S_3] &= 1 \\
\mathbf{E}[S_2] &= \left(\frac{1-q}{p_3}\right)\mathbf{E}[S_3] + q \\
&= (1-q)e^{\lambda w_3} + q \\
\mathbf{E}[S_1] &= \left(\frac{1-q}{p_2}\right)\mathbf{E}[S_2] + q \\
&= (1-q)^2 e^{\lambda(w_2+w_3)} + q(1-q)e^{\lambda w_2} + q
\end{aligned}$$

Without loss of generality, assume the job length is 1. Then the expected job completion time is given by

$$\begin{aligned}
T(\boldsymbol{w}) &= \sum_{i=1}^{3} (\mathbf{E}[S_i] \cdot \mathbf{E}[W_i]) \\
&= \frac{q}{\lambda}\left[\left(e^{\lambda w_1} + qe^{\lambda w_2} + e^{\lambda w_3}\right) + (1-q)\left(e^{\lambda(w_1+w_2)} + e^{\lambda(w_2+w_3)}\right)\right] \\
&\quad + \frac{1}{\lambda}\left[(1-q)^2 e^{\lambda} - 2q - 1\right]
\end{aligned}$$

By the symmetry property shown in Theorem 4.28, there exists an optimal length vector $\boldsymbol{w}^* = (w^*, 1-2w^*, w^*)$. The expected job completion time with $\boldsymbol{w}^*$ is

$$\begin{aligned}
T(\boldsymbol{w}^*) &= \frac{q}{\lambda}\left[2e^{\lambda w^*} + qe^{\lambda(1-2w^*)} + 2(1-q)e^{\lambda(1-w^*)}\right] \\
&\quad + \frac{1}{\lambda}\left[(1-q)^2 e^{\lambda} - 2q - 1\right]
\end{aligned}$$

Lemma 4.29 *Let $\boldsymbol{w}^* = (w^*, 1-2w^*, w^*)$ be an optimal length vector that minimizes the expected job completion time T. Then w^* is monotone decreasing in q over the interval $(0, 1)$. $w^* = \frac{1}{3}$ when $q = 1$, and w^* approaches $\frac{1}{2}$ when q approaches 0.*

Proof. Taking the derivative of $T(\boldsymbol{w}^*)$, re-arranging terms, and equating it to zero gives

$$h(w^*, q) = e^{3\lambda w^*} - (1-q)e^{\lambda(1+w^*)} - qe^{\lambda} = 0 \tag{4.52}$$

From (4.52), it is straightforward that $w^* = 1/3$ when $q = 1$, and w^* approaches $1/2$ when q approaches 0. We show the monotonicity property as follows. When $q \neq 0$, $qe^{\lambda} > 0$, and thus from (4.52) we have

$$e^{3\lambda w^*} - (1-q)e^{\lambda(1+w^*)} > 0 \tag{4.53}$$

Taking partial derivative of $h(w^*, q)$ with respect to w^* gives

$$\frac{\partial h(w^*)}{\partial w^*} = 3\lambda e^{3\lambda w^*} - (1-q)\lambda e^{\lambda(1+w^*)} \tag{4.54}$$

Combining (4.53) and (4.54), we have

$$\frac{\partial h(w^*)}{\partial w^*} > 0 \tag{4.55}$$

Taking partial derivative of $h(w^*, q)$ with respect to q gives

$$\frac{\partial h(w^*)}{\partial q} = e^{\lambda(1+w^*)} - e^{\lambda} > 0, \tag{4.56}$$

where the inequality follows from the fact that exponential function is monotone increasing and $w^* > 0$. Combining (4.55) and (4.56), we know that $h(w^*, q)$ is monotone increasing in w^* and q. Therefore, when q increases, to make the equality $h(w^*, q) = 0$ hold, w^* must decrease. Thus, w^* is monotone decreasing in q. $\square$

The following theorem follows from Lemma 4.29.

Theorem 4.30 *Let $\mathbf{w}^* = (w_1^*, w_2^*, w_3^*)$ be an optimal length vector that minimizes the expected job completion time. If $0 < q < 1$, then $w_1^* = w_3^*$ and $w_1^* > w_2^*$.*

4.9 Uniform Poisson Checkpointing Systems

In practice, dynamically deciding optimal checkpoint placement is a complex task. Therefore, many systems make checkpoints periodically over equidistant intervals due to the simplicity of this strategy. In this case, all tasks have the same length, which is equal to the checkpoint interval. We define the following system to capture such equidistant checkpoint placement.

Definition 4.9 (Uniform Poisson Checkpointing System) *A* uniform Poisson checkpointing system *is a Poisson checkpointing system where all tasks are uniform.*

When the job length is fixed and tasks have the same length, one wishes to decide the optimum task length to minimize the expected job compeltion time. If the cost for generating a checkpoint is negligible, it would be plausible to make the interval as small as possible. Usually creating a checkpoint introduces some amount of cost and thus the total overhead can be significant if the checkpoint interval is made arbitrarily small. Therefore, different from the analysis in Section 4.8, we assume a positive cost for generating a checkpoint.

Let L be the job length, w be the task length, g be the cost for generating a checkpoint, and λ be the task failure rate. Then the mean time between failures is $M = 1/\lambda$. We show a first order approximation to the optimum checkpoint interval.

4.9.1 A First Order Approximation to the Optimum

It has been shown that $w = \sqrt{2gM}$ is a first order approximation to the optimum reliable checkpoint interval [76], under the assumption that $g \ll M$. We extend this result to unreliable checkpoints. Our result in summarized in Theorem 4.31.

Theorem 4.31 *Let w be the task length and p be the task success probability. Let g be the cost of generating a checkpoint and q be the checkpoint reliability. Let M be the mean time between failures. Assuming $g \ll M$ and $p+q > 1$, a first order approximation to the optimum checkpoint interval is*

$$w = \sqrt{\frac{2q}{2-q} gM}$$

Proof. The number of tasks is $n = L/w$. Generating a checkpoint after a successful task run costs g. Therefore, repeatedly running a single task until the checkpoint is successfully created takes expected time

$$k = \frac{1/p - 1}{\lambda} + g, \tag{4.57}$$

where $p = e^{-\lambda w}$. Because $p + q > 1$, by Theorem 4.13, we have

$$T = k\left[\frac{qn}{1-r} + \Theta(1)\right] \approx \frac{kqn}{1-r}, \tag{4.58}$$

where $r = (1-q)/p < 1$. Combining (4.57) and (4.58) gives

$$\begin{aligned}
T &\approx \left(\frac{1/p-1}{\lambda} + g\right)\frac{qn}{1-r} \\
&= \left(\frac{1/p-1}{\lambda} + g\right)\frac{pqn}{p+q-1} \\
&= \left(\frac{e^{\lambda w}-1}{\lambda} + g\right)\left(\frac{qe^{-\lambda w}}{e^{-\lambda w}+q-1}\right)\cdot\frac{L}{w} \\
&= \left(\frac{1+(\lambda g-1)e^{-\lambda w}}{e^{-\lambda w}+q-1}\right)\cdot\frac{1}{w}\cdot\frac{qL}{\lambda}
\end{aligned}$$

Taking derivative with respect to w and equating the result to zero gives

$$(\lambda g - 1) + e^{\lambda w}\left[(\lambda g-1)(q-1)(1+\lambda w) + 1 - \lambda w\right] + e^{2\lambda w}(q-1) = 0 \tag{4.59}$$

Use the expansion of exponential function as far as the second degree term and ignore higher degree terms. Then (4.59) becomes

$$\begin{aligned}
&\frac{\lambda}{2}[(3\lambda g+1)(1-q)+1]w^2 + 2\lambda g(1-q)w - qg \\
=\ &\frac{1}{2M}\left[\left(\frac{3g}{M}+1\right)(1-q)+1\right]w^2 + \frac{2g}{M}(1-q)w - qg \\
=\ &0
\end{aligned}$$

Because $g \ll M$, we neglect the term (g/M) and simplify the expression to

$$\frac{2-q}{2M} w^2 = qg$$

Solving for w gives

$$w = \sqrt{\frac{2q}{2-q} gM}$$

$\square$

We have several observations on this formula. The first observation is that w is determined only by the reliability of checkpoints q, the cost for making a checkpoint g, and the mean time between failures M. Second, note that for $q = 1$ (i.e., totally reliable checkpointing), our result reduces to $w = \sqrt{2gM}$, in agreement with the result in [76]. Third, as $2q/(2-q)$ is monotone increasing in $[0, 1]$, our result indicates that checkpointing should become more frequent with decreasing checkpoint reliability.

4.9.2 Feasible Region of the Approximation

The approximation $w = \sqrt{\frac{2q}{2-q} gM}$ in Theorem 4.31 requires $p + q > 1$. Therefore, there is a feasible region of q to guarantee this condition. Lemma 4.32 summarizes this result.

Lemma 4.32 *To guarantee $p + q > 1$ when $g \ll M$ and $w = \sqrt{\frac{2q}{2-q} gM}$, it is sufficient that*

$$1 - \sqrt{1 - \frac{2g}{M}} < q \leq 1$$

Proof. Combining $p = e^{-\lambda w}$, $\lambda = 1/M$ gives

$$\begin{aligned} p + q &= e^{-\lambda w} + q \\ &= e^{-w/M} + q \\ &\geq 1 - \frac{w}{m} + q \end{aligned}$$

Therefore, to guarantee $p + q > 1$, it is sufficient that

$$q > \frac{w}{m} \tag{4.60}$$

Plugging $w = \sqrt{\frac{2q}{2-q} gM}$ into (4.60) gives

$$q > \sqrt{\frac{2qg}{(2-q)M}}$$

Solving the inequality for q gives

$$1 - \sqrt{1 - \frac{2g}{M}} < q < 1 + \sqrt{1 - \frac{2g}{M}}$$

Being a probability, q never exceeds 1 and thus we have

$$1 - \sqrt{1 - \frac{2g}{M}} < q \leq 1$$

□

When the cost g for generating a checkpoint is sufficiently small compared to the mean failure time M, their ratio (g/M) tends to be fairly small, and thus $(1 - \sqrt{1 - 2g/M})$ is close to 0. Therefore, Lemma 4.32 indicates that q has a wide feasible region and thus Theorem 4.31 applies to a wide range of uniform Poisson checkpointing systems.

Chapter 5

Conclusion

In this thesis, we presented models and algorithms for two aspects of efficient fault-tolerant cloud computing infrastructure, task assignment and checkpoint placement. The Hadoop task assignment model provides a tool to compare the efficiency of different task assignment strategies. The flow-based algorithm achieves good approximation to optimum assignments. Witness graphs and token production systems provide an approach to analyze online task assignments. The extended checkpoint model offers a way of evaluating checkpoint placements and configurations when checkpoint unreliability becomes significant. While most of the work was initially motivated by challenges arising from cloud computing, the models and techniques developed are potentially applicable to other systems.

Cloud computing has opened up many opportunities in both research and business. It has also presented many interesting technical challenges. The approach we take is to formulate practical problems arising from cloud computing in ways that make them amenable to theoretical treatment. Such a rigorous and mathematical approach leads to many insights that help one better understand and design cloud computing infrastructure. We are confident that our research approach provides a promising and crucial voice to this new frontier of large-scale computing.

Bibliography

[1] P. Abdulla and R. Mayr. Minimal cost reachability/coverability in priced timed Petri nets. *Foundations of Software Science and Computational Structures*, pages 348–363, 2009.

[2] P. Abdulla and R. Mayr. Computing optimal coverability costs in priced timed Petri nets. In *26th Annual IEEE Symposium on Logic in Computer Science*, pages 399–408. IEEE, 2011.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB*, 2(1):922–933, 2009.

[4] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM*, 44(3):486–504, 1997.

[5] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[6] Y. Azar, J. S. Naor, and R. Rom. The competitiveness of on-line assignments. *Journal of Algorithms*, 18(2):221–237, 1995.

[7] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. Roux. Comparison of different semantics for time Petri nets. *Automated Technology for Verification and Analysis*, pages 293–307, 2005.

[8] K. Birman, G. Chockler, and R. van Renesse. Towards a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[9] D. Borthakur. The Hadoop distributed file system: Architecture and design. `http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf`, 2008.

[10] E. Bortnikov. Open-source grid technologies for web-scale computing. *SIGACT News*, 40(2):87–93, 2009.

[11] M.-S. Bouguerra, D. Trystram, and F. Wagner. Complexity analysis of checkpoint scheduling with variable costs. *IEEE Transactions on Computers*, 2012.

[12] N. Buchbinder and J. Naor. Fair online load balancing. In *Proceedings of the eighteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 291–298. ACM, 2006.

[13] R. E. Burkard. Assignment problems: Recent solution methods and applications. In *System Modelling and Optimization: Proceedings of the 12th IFIP Conference, Budapest, Hungary, September 2-6, 1985*, pages 153–169. Springer, 1986.

[14] K. M. Chandy. A survey of analytic models of rollback and recovery strategies. *Computer*, 8(5):40–47, 1975.

[15] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 100(6):546–556, 1972.

[16] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng. Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected

graphs. In *Proceedings of the 43rd annual ACM Symposium on Theory of Computing*, pages 273–282. ACM, 2011.

[17] R. Cole, K. Ost, and S. Schirra. Edge-coloring bipartite multigraphs in $O(E \log D)$ time. *Combinatorica*, 21(1):5–12, 2001.

[18] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms, 2nd ed.* MIT press Cambridge, MA, 2001.

[19] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[20] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell B. E. architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[21] F. G. de Lima Kastensmidt, G. Neuberger, R. F. Hentschke, L. Carro, and R. Reis. Designing fault-tolerant techniques for SRAM-based FPGAs. *IEEE Design & Test of Computers*, 21(6):552–562, 2004.

[22] J. Dean. Experiences with MapReduce, an abstraction for large-scale computation. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. ACM New York, NY, USA, 2006.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA*, pages 137–150, 2004.

[24] P. Doyle and J. Snell. *Random walks and electric networks*. Carus mathematical monographs. Mathematical Association of America, 1984.

[25] A. Duda. Effects of checkpointing on program execution time. *Information Processing Letters*, 16(5):221–229, 1983.

[26] J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[27] M. J. Fischer. Efficiency of equivalence algorithms. In *Complexity of Computer Computations*, pages 153–167, 1972.

[28] M. J. Fischer, X. Su, and Y. Yin. Assgining tasks for efficiency in Hadoop. *Technical Report, YALEU/DCS/TR-1423, March 31, 2010*, 2010.

[29] M. J. Fischer, X. Su, and Y. Yin. Assigning tasks for efficiency in Hadoop. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 30–39. ACM, 2010.

[30] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[31] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB*, 2(2):1402–1413, 2009.

[32] H. N. Gabow and O. Kariv. Algorithms for edge coloring bipartite graphs. In *Proceedings of the tenth annual ACM Symposium on Theory of Computing*, pages 184–192. ACM, 1978.

[33] B. Galler and M. J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co., San Francisco, 1979.

[35] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):252–261, 1996.

[36] E. Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, 26(2):259–270, 1979.

[37] A. Goel, M. Kapralov, and S. Khanna. Perfect matchings via uniform sampling in regular bipartite graphs. *ACM Transactions on Algorithms*, 6(2):27, 2010.

[38] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the Symposium on Security and Privacy*, pages 154–165. IEEE, 2003.

[39] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[40] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, pages 416–429, 1969.

[41] D. Griffeath, J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. Graduate Texts in Mathematics. Springer, 1976.

[42] M. Hack. *Decidability questions for Petri nets*. Garland Publishing, New York, 1975.

[43] Hadapt: The adaptive analytical platform for big data. `http://www.hadapt.com/`.

[44] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10(1):26–30, 1935.

[45] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM New York, NY, USA, 2008.

[46] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of Computing*, pages 302–311. ACM, 1984.

[47] R. Karp and R. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[48] R. Keller. A fundamental theorem of asynchronous parallel computation. In *Parallel Processing*, pages 102–112. Springer, 1975.

[49] T. Koopmans. Analysis of production as an efficient combination of activities. *Activity Analysis of Production and Allocation*, 13:33–37, 1951.

[50] S. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the fourteenth annual ACM Symposium on Theory of Computing*, pages 267–281. ACM, 1982.

[51] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics*, 52(1), 2005. Originally appeared in *Naval Research Logistics Quarterly,* 2, 1955, 83–97.

[52] V. Kulkarni, V. Nicola, and K. Trivedi. Effects of checkpointing and queueing on program performance. *Stochastic Models*, 6(4):615–648, 1990.

[53] J. Lambert. A structure to decide reachability in Petri nets. *Theoretical Computer Science*, 99(1):79–104, 1992.

[54] R. Lämmel. Google's MapReduce programming model—Revisited. *Science of Computer Programming*, 68(3):208–237, 2007.

[55] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1):259–271, 1990.

[56] W. Leontief. *The Structure of American Economy*. Harvard University Press, 1941.

[57] L. Lin, V. Lychagina, and M. Wong. Tenzing – a SQL implementation on the MapReduce framework. *Proceedings of the VLDB*, 4(12):1318–1327, 2011.

[58] E. Mayr. An algorithm for the general Petri net reachability problem. In *Proceedings of the thirteenth annual ACM Symposium on Theory of Computing*, pages 238–246. ACM, 1981.

[59] M. Mitzenmacher. *The power of two choices in randomized load balancing*. PhD thesis, University of California, Berkeley, 1996.

[60] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, volume 1, pages 255–312, 2001.

[61] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós. On scheduling in map-reduce and flow-shops. In *Proceedings of the 23nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 289–298. ACM, 2011.

[62] V. F. Nicola. Checkpointing and the modeling of program execution time. In M. R. Lyu, editor, *Software Fault Tolerance*. John Wiley & Sons, Inc., 1994.

[63] Oracle loader for Hadoop. http://www.oracle.com/technetwork/bdc/hadoop-loader/overview/index.html.

[64] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD*, pages 1099–1110, 2008.

[65] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio. Distribution-free checkpoint placement algorithms based on min-max principle. *IEEE Transactions on Dependable and Secure Computing*, 3(2):130–140, 2006.

[66] Quest data connector for Oracle and Hadoop. http://www.quest.com/data-connector-for-oracle-and-hadoop/.

[67] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE, 2007.

[68] M. Ronald and B. Peter. Input-output analysis: Foundations and extensions, 2nd edition. *Cambridge University Press*, 2009.

[69] V. V. Ruiz, D. de Frutos Escrig, and F. C. Gomez. On non-decidability of reachability for timed-arc Petri nets. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models*, pages 188–196, 1999.

[70] G. Sacerdote and R. Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In *Proceedings of the ninth annual ACM Symposium on Theory of Computing*, pages 61–76. ACM, 1977.

[71] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[72] Apache Sqoop. http://sqoop.apache.org/.

[73] X. Su and G. Swart. Oracle in-database Hadoop: when MapReduce meets RDBMS. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 779–790. ACM, 2012.

[74] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB*, 2(2):1626–1629, 2009.

[75] S. Toueg and Ö. Babaoglu. On the optimum checkpoint selection problem. *SIAM Journal on Computing*, 13(3):630–649, 1984.

[76] J. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[77] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation, San Diego, CA*, 2008.