Master Thesis

April 5, 2017

BPM

Discrete Event Simulation for Optimal Role Resolution in Workflow Processes

Filip Kočovski

of Lugano, Switzerland (10-932-994)

supervised by

Prof. Dr. Daning Hu

Dr. Markus Uhr

University of Zurich[UZH]

ifi

Master Thesis

BPM

Discrete Event Simulation for Optimal Role Resolution in Workflow Processes

Filip Kočovski

University of Zurich[UZH]

ifi

Master Thesis

Author: Filip Kočovski, filip.kocovski@uzh.ch

Project period: November 15, 2016 - May 15, 2017

Business Intelligence Research Group
Department of Informatics, University of Zurich

Acknowledgements

Abstract

Zusammenfassung

Contents

List of Figures

List of Tables

List of Listings

Structure for the thesis adapted from https://wwz.unibas.ch/fileadmin/wwz/redaktion/fmgt/Images/FinanzmanagementLeitfadenfuerArbeiten.pdf

Chapter 1

Introduction

1.1 Problem Definition

Workflows are IT solutions that can help increase efficiency and get tasks done better and faster. However a key element of each workflow process still remains the human aspect. This human aspect can take many facets, such as humans analyzing a process, humans designing a process and humans executing the latter. This thesis focuses on the latter *i.e.*, where human agents interact with the workflow process in order to work on tasks. A business process that has been efficiently analyzed and subsequently optimally implemented still cannot ensure optimal execution, or no optimal execution can be achieved while a human intervention for task execution is present. It is here that optimal role resolution comes in play: optimally choosing and assigning a specific task inside the workflow process to the best possible actor is a non trivial task that has to be solved in order to close the "optimization" circle that workflow engines advertise.

This field is relevant since an optimal role resolution can bring optimization from many sides: 1. cost savings, 2. fairness in workload assignment 3. optimal resources usage.

Currently many different workflow engines exist, ranging from complete fully functional suites and down to extensible frameworks that allow the implementer to adapt it to its own needs. However all these solutions lack optimality in the task assignment sector.

1.2 Objectives

The objectives of these thesis build upon the work of Zeng and Zhao [39], in which they depicted preliminary policies for optimal role resolution, and extends these capabilities from a threefold perspective: 1. further develops the mathematical premises and extends the capabilities of the batching policies proposed by Zeng and Zhao 2. explores the capabilities offered by reinforcement learning as addition and improvement for even precises, faster and better task assignment 3. deployment of the aforementioned optimization techniques in an operative environment of a real estate company using a workflow engine.

Formally, this thesis tries to answer the following research questions:

1. Are there better optimization techniques for optimal role resolution techniques inside workflow processes?
2. Is the deployment of optimization policies in a working environment for a workflow engine a critical success factor?
3. How is optimization in the field of task assignment perceived by the workflow users (actors)?

1.3 Thesis Structure

This thesis is subdivided in five main chapters:

- Chapter 1 gives an overview of why the chosen topic is relevant, what is the current context of the work and how this work fits in. It moreover articulates the central research questions that permeate this thesis and gives an overview of this essay
- Chapter 2 gives an overview of the most important conceptual definitions and the state of the art literature review in the touched thematic topics of this work. Conclusively this chapter critically reflects upon the existing literature and exposes the deficits that this thesis aims filling
- Chapter 3 gives an overview of the approach used for the research *e.g.*, the analysis environment and the used tools, states the hypothesis that wants to be proved and eventually describes statistically and qualitatively the data sets upon which the methodology is applied
- Chapter 4 builds upon Chapter 3 and makes its way into the hypothesis test field and the respective analysis results. Furthermore looks introspectively on the data correlation and gives an interpretation of the latter. Eventually in this section a statement about the contribution that the results bring into this field is given
- Chapter 5 is the culminating chapter in which a summary of the key findings of the thesis are outlined, the research questions posed in Section 1.2 are answered by looking at the actual usability, limitations and to whom the results are most applicable. Finally outlooks about the future trends and how the empirical results of this thesis can be extended by prospective researchers.

Chapter 2

Theoretical Foundations

2.1 Literature Overview

This section servers as an overview of the state of the art literature that exists and has been used as a foundation basis for this work. Section 2.1 is divided in different thematic subsections.

2.1.1 Queueing

Queuing is a topic that talks about how people or more general agents are to be server while waiting.

Starting with one of the most notable contributions to this field done by Kendall in 1953 and his work on the Markov chains in queuing theory, where he formally defines different types of queues [17].

In 2016, Adan describes the necessary basic concepts for queuing theory and an important topic here is the statistical foundation outlined in his work about different modeling techniques for randomized generation rates, such as the Erlang's distributions [1].

Pinedo outlines in his work in 2008 the most prominent key metrics that can be used in order to assert and measure queues performance [24].

Sun and Zhao in their work cover the aspect of formal analysis for workflow models and they claim that it should help "...alleviating the intellectual challenge faced by business analysts when creating workflow models" [32].

2.1.2 Workflow

A good starting point in the workflow thematic is Macintosh's work in which he gives an overview of the five levels of process maturity [20]:

1. Initial, the process has to be set up
2. Repeatable, the process has to be repeatable
3. Defined, documentation standardization of processes
4. Managed, measurement and control of processes
5. Optimizing, continuous process improvement

Even though Georgakopoulos' work dates back to 1995, he still gives a comprehensive business oriented overview of the different workflow technologies present on the market [12].

On this note, Giaglis lays out four different process perspectives: 1. Functional 2. Behavioral 3. Organizational 4. Informational

His framework focuses on three dimensions: 1. Breadth, where modeling goals are typically addressed by technique 2. Depth, where modeling perspectives are covered 3. Fit, where typical project to which techniques can be fit

The presented framework is used to combine the three different dimensions in order to assert a possible best fit of a specific modeling technique based on which approach to be used under the constraints of a modeling perspective to cover [15].

Mentzas focuses on a qualitative level on how workflow technologies can facilitate implementation of business processes by focusing on the pros and cons of adopting alternative workflow modeling techniques [22]. Moreover he formally defines what a workflow management system is and subdivides it in three main categories: 1. Process modeling 2. Process re-engineering 3. Workflow implementation and automation

Each level of maturity as defined by Macintosh requires a different model, such as the first three levels might require more descriptive models whereas levels four and five require decision support keen models in order to monitor and control processes [22].

Aguilar describes the main modeling techniques existing with workflow being one of them [2].

The key core topics on which this thesis lays its foundations upon is the work done by Zeng in 2005. Effective role resolution *i.e.*, the mechanism of assigning tasks to individual workers at runtime according to the role qualification defined in the workflow model [39], is the core aspect that is being extended during this thesis work.

Zeng differentiates between staffing decisions and role resolution, with the former being the assignment one or more role to each user and the latter being the assignment of a specific task to an appropriate worker at runtime [39]. Staffing decisions are usually made off-line and periodically, thus being more of a strategic nature [39]. If role resolution were to be made on-line it could translate to a major operational level decision *i.e.*, the differentiation between strategic vs. operational playing role [39].

He moreover defines three roles a workflow can fulfill: 1. System built-in policies 2. User customizable policies 3. Rule based policies

Considering capacities of resources restrictions under the assignment problem is an NP-hard computational problem and in his work Zeng focuses on how to solve the assignment problem and scheduling decisions with consideration of worker's preferences [39]. For this purpose he defines five workflow resolution policies:

1. Load balanced approach (LLQP)
2. Shared queue (SQ)
3. K-Batch
4. K-Batch-1
5. 1-Batch-1

For all batching policies a simplified version of the dynamic minimization of the maximum flowtime (DMF) has to be solved [39].

Zeng's key findings are outlined as follows: 1. Batching policies to be used when system load is medium to high 2. Processing time variation has major impact on system performance *i.e.*, higher variation favors optimization based policies 3. Average workload and workload variation can be significantly reduced by online optimization 4. 1-Batch-1 online optimization policy yields best results in operational conditions

Interestingly enough, workflow implementation in real world cases is not always only coupled with directly measurable effects, sometimes even unexpected results happen. What is called the "workflow paradox" according to Reijers is the fact that the very fact of companies accepting requests for workflow introduction might actually be the most promising way that leads to potentially better and more suitable alternatives [26].

Specifically speaking on the data flow inside workflow processes, one has to consider possible anomalies that might happens. This has been extensively studies by Sun *et al.* where they formally define data flow methodologies for detecting such anomalies [33]. Their framework is divided in two components: 1. Data flow specification 2. Data flow analysis

Yet again we stumble upon mentioning that simulation for workflow management systems is usually inefficient and inaccurate [33]. They moreover discuss aspects that data requirements have been analyzed but the required methodologies on discovering data flow errors have not been extensively researched [33].

A more recent taxonomy of different BPM application is given by a collaboration between SAP and accenture in 2009 [10].

In the realm of workflow processes and engines BPMN's notation permeates the field and the work of Silver summarizes these foundations very well [27].

An analysis of the critical success factors (CSF) for BPM is required in order to assert a product validity and this has been done by Trkman where he defines CSF from three perspectives [36]: 1. Contingency theory 2. Dynamic capabilities 3. Task-technology fit theory

Change management in workflow is yet another interesting aspect that should be considered and this has been broadly studied by Wang where he developed an analytical framework for workflow change management through formal modeling of workflow constraints [37].

In companies different types of workflow models can exist and Fan focuses on two of these, namely: 1. Conceptual 2. Logical

Conceptual models serve as documentation for generic process requirements whereas logical models are used as definitions for technology oriented requirements [11]. One difficult aspect is the transition from the former to the latter and Fan proposes a formal approach to efficiently support such transitions [11].

2.1.3 Reinforcement Learning

Reinforcement learning is a branch of machine learning that promises to overcome the drawbacks posed by the latter by not requiring a training set for efficient machine decisions.

One of the first Monte Carlo based policy gradient methods is the algorithm proposed by Williams called REINFORCE [38].

Policy gradient methods with value function approximation and their convergence is of vital importance and this can be achieved by representing the policy by an own function approximation which is independent of the value function and it is updated according to gradient of the expected rewards with respect to the afore mentioned policy [35].

Discretization of the state action space is not always feasible and different techniques have to be used for tractability. Smith proposes such an approach which he calls "self-organizing map" [30].

As Markov Decision Processes grow in size, so does the required computational memory to solve possible discrete lookup tables modeling the state-actions spaces that characterizes them. Notable examples that show how large some of the most common problems can be: 1. the game of backgammon has a total of 10^{20} states 2. the traditional Chinese abstract board game Go has an estimated total of 10^{170} states 3. flying a helicopter or having a robot move in space all require a continuous state space.

This huge state space requirement is a clear limitation to lookup tables. Even if memory would not be a constraint, the actual learning from such tables would be infeasible. In order to pragmatically learn by reinforcement on such huge problems, value function approximation in the domain of reinforcement learning proves to be a viable solution. For different types of reinforcement learning approaches *i.e.*, Monte-Carlo (MC) or Temporal Difference (TD) methods exist different types of value function approximation, ranging from simple linear combinations of features for MC to neural networks for TD learning. All these different methodologies are outlined in the tutorial by Geramifard [13].

When working with on-line algorithms such as TD(0) it is important to choose correct parameters for an effective learning process, otherwise the learning algorithm put in place might never converge towards an optimal solution. This aspect is being discussed by Korda in which he depicts different non-asymptotic bounds for the temporal difference learning algorithms [18].

There are two main fields in reinforcement learning, one is using value function approximation for either the state value function or for using control mechanisms with the state action value function, while the other one is using policy gradient methods for policy optimization. The latter offers different methods such as the naive finite difference methods, monte carlo based policy gradient methods and finally actor critic policy gradient methods [29].

Notable works in the field of reinforcement learning and its application include Google DeepMind work on novel algorithms for tackling fields previously barely scratched, as mentioned by Mnih *et al.* and Silver *et al.* [23, 28].

Sutton started working on the reinforcement learning topic in the early nineties and is now planning his third edition of his famous book on machine learning, which is due in 2017 [34]. In our case reinforcement learning is used in order for the policies to be able to alone get better by continuously analyzing their own decision models and optimize upon them.

2.1.4 Optimization

For all batching policies implemented in this work, a mixed integer optimization was solved in order to optimally assign jobs to users in the workflow processes. The generalized assignment problem is a very well known problem in combinatorial mathematics. Cattrysse gives an overview of different algorithms for solving the generalized assignment problem [8]. Heuristics are also a viable solution for solving such adaptation of the generalized assignment problem, as Racer states [25]. Moreover a global perspective of optimization from a mathematical perspective is given in Boyd's work on convex optimization [7].

Last but not least, according to the AIMMS guidelines, there are different linear programming tricks that can be used to shape such problems in solvable outlines [6]. In this thesis, a specific linear programming trick, called either-or constraints, was used by adding so called auxiliary variables to the evaluation method presented in order to efficiently solve an otherwise non solvable equation [6, p. 77].

2.1.5 Simulation

Simulating queues can prove to be extremely difficult. The main differentiation needed here is that between continuous and step functions: the former is the result when the events being simulated yield values that if plotted against the simulation time give a continuous function. On the other hand, if we simulate events that yield discrete values, such as inventory changes in a storage facility and plot the results against the simulation time we would get so called step functions [21].

According to Matloff, there exist different world views for discrete event programing, as he calls them paradigms [21]:

1. Activity oriented
2. Event oriented
3. Process oriented

Activity oriented can be summarized as simulation events where time is being subdivided in tiny intervals at which the program checks the status for all simulated entities. Since petite subdivisions of time are possible in such types of simulations, it is clear that the program might prove extremely inefficient, since most of the time there won't be any change in state for the simulated entities [21]. Event oriented circumnavigate this issue by advancing the simulation time directly to the next event to be simulated. By filling these gaps, a dramatical increase in computation can be observed [21]. Last but not least, the process oriented simulation models each simulation activity as a process or thread. Management of threads has steadily decreased in todays computation since many different packages for governing such tasks.

On another note, Bahouth focuses in work on algorithmic analysis of discrete event simulation supplemented with focus on factors such as compiler efficiency, code interpretation and caching memory issues [3]. According to his findings, a significant speedup can be achieved if one addresses the afore mentioned facets.

2.2 Research Deficit

Chapter 3

Methodology

3.1 Analysis Structure

3.1.1 Tools

Different tools were used in the analysis environment in order to efficiently simulate and analyze the work of this thesis.

The whole architecture is subdivided as follows:

1. The simulation environment is based on `Python 3.5.2`[1] using the `Anaconda`[2] platform and as a discrete event simulation the `SimPy 3.0.10`[3] package is used.
2. The resulting data are interpreted and analyzed using Python and its plotting library: `Matplotlib 2.0.0`[4].
3. `Tensorflow 1.0`[5] is the library used for the neural networks modeling.
4. Coding was done using `PyCharm 2017.1`[6] as IDE for Python.
5. For solving the mixed integer problems for batching policies `Gurobi 7.0.1`[7] was used.

3.1.2 Discrete event simulation using SimPy

SimPy is a Python process-based discrete-event simulation framework. It exploits Python generators according to which it models its processes.

Active components such as agents in a workflow are modeled as processes which live inside an environment and the interaction between them happens via events.

As previously mentioned, processes in SimPy are described by Python generators. During their lifetime they create events yield (Note that with the term `yield` here it is to be understood as Python's yield statements)[8] them to the environment, which then wait until they are triggered.

[1] https://www.python.org (accessed: 06.01.2017)
[2] https://www.continuum.io/anaconda-overview (accessed: 03.04.2017)
[3] https://simpy.readthedocs.io/en/latest/ (accessed: 06.01.2017)
[4] http://matplotlib.org/ (accessed: 03.04.2017)
[5] https://www.tensorflow.org/ (accessed: 03.04.2017)
[6] https://www.jetbrains.com/pycharm/ (accessed: 03.04.2017)
[7] http://www.gurobi.com (accessed: 06.01.2017)
[8] https://docs.python.org/3.5/reference/simple_stmts.html#the-yield-statement (accessed: 06.01.2017)

The important logic to understand here is how SimPy treats yielded events: when a process yields an event it gets suspended. From the suspended state a process gets resumed when the event actually occurs (or in SimPy's notation when it gets triggered).

SimPy offers a built-in event type called `Timeout`: events of this type are automatically triggered after a determined simulation time step. Consistency is asserted since a timeout event are created and called by called the appropriate method of the passed `Environment`.

3.1.3 Analysis Environment

The analysis environment consists in an object-oriented implementations of workflow process elements such as user task, starting, decision and end nodes which have been developed to allow the simulation framework to effectively run. This object-oriented exoskeleton implementation of the workflow elements can be seen depicted in Figure 3.1.

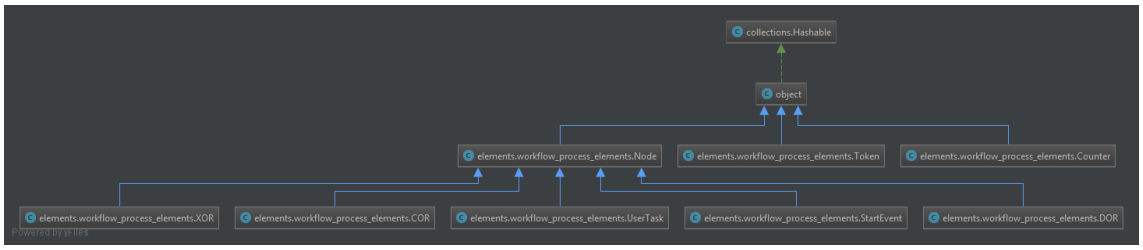

Figure 3.1: Workflow elements

The core elements of a workflow process (relevant for the simulation environment) are start nodes, user tasks, decision nodes and end nodes. Start events are used to indicate where and how a process starts and usually each process has only one such event [27, p. 42]. No distinction between trigger types is being made.

Start event

Start event objects require a simulation environment, a generation interval, an actions to follow array and its corresponding weights. The generation interval is generated in a three step process: 1. before the simulation starts, a fixed service interval time unit s, number of users n and an average system load l are set. In contrast to Zeng's and Zhao's work, where the generation λ interval follows a Poisson distribution [39] and is defined as shown in Equation 3.1, here the generation interval is a plain scalar value. 2. for a Poisson random exponential sampling of the generation rate, `NumPy`'s implementation of its exponential distribution is used[9].

$$\lambda = \frac{ln}{s} \tag{3.1}$$

The actions to be followed are also defined in a two step process: 1. a per workflow process action pool is defined a priori in order to assert that tokens navigate the process in a "semantically correct" fashion 2. then a weights vector is defined which assigns a weight to each possible action path to the actions pool.

[9] https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.exponential.html (accessed: 06.01.2017)

Such an approach allows to fine tune how often tokens will follow a predefined path along the process in order to efficiently simulate and put under stress specific paths of the process.

In order to assert fairness among all simulation runs a master random state is assigned to the start event. This master random state is generated from the PCG family of random generators which exhibit peculiar characteristics, one amongst all is the possibility of "jumping ahead" in the state. By such means of ahead jumps it is possible to assign a fixed number of random yet consistent choices among all runs, since each generated token receives from the start event a "jumped" copy from the master state. For a better overview of the characteristics of the PCG random generators family consult its official outline [10].

Even though tokens are generated infinitely, this process is controlled from the simulation environment where a discrete simulation time steps have to be set, as it can be seen from Listing 3.1.

This can be interpreted as that the whole simulation will persist for 100 time steps and it will then stop when the internal clock reaches 100. Please note that events that have been scheduled for time step 100 will not be processed. The logic is similar to a new environment where the clock is zero and no event have been processed yet.

```
# "global" variables
SIM_TIME = 100
...
# runs simulation
env.run(until=SIM_TIME)
```

Listing 3.1: Starting the simulation with discrete time steps

User task

User task objects also require a simulation environment, a policy, a descriptive name, a service interval and task variability. Each user task has a unique `child` field which is being set prior to starting the simulation.

In regards to parameters service interval and task variability a detailed explanation is required. Both are used to randomly sample service rate intervals for each user active during the simulation. Zeng and Zhao in their work follow a two way process to generate such intervals [39, p. 8]. However in this thesis' implementation a refined version of this process is used: 1. at initialization time, each user task receives a service rate s and a task variability t value 2. inside the policy request method, for each user task a sample of an average processing time following an Erlang distribution (a special case of the gamma distribution) which takes as input parameters a shape k and a scale θ is made. The shape value k, as the name suggests, defines the curve shape that the Erlang distribution will follow. In this case both values k and θ are dynamically evaluated at runtime as $k = s/t$ and $\theta = t$. This concept is depicted in Listing 3.3 3. the average processing time becomes a unique value of each user task object and is used by each policy to sample each user's service time, again from an Erlang sampled pool as depicted in Listing 3.2 and we shall call this value p_j

Listing 3.2 gives a glimpse of the inner logic of how policies work. It is however out of scope for this section to cover this aspect and it is provided "as is". For each user eligible to work the assigned token, its service rate is sampled following the Erlang distribution. This time, the Erlang distribution takes as parameters the unique average processing time p_j of user task j and a value worker variability, which is a unique property of each policy, which we shall call w.

In order to sample a service rate p_{ij} following the Erlang distribution for each user i, shape k is evaluated as $k = p_j/w$ and scale as $\theta = w$ as it can be seen in Listing 3.2

[10] http://www.pcg-random.org/ (accessed: 03.04.2017)

```
def request(self, user_task,token):
    average_processing_time = token.random_state.gamma(
        user_task.service_interval ** 2 / user_task.task_variability,
        user_task.task_variability / user_task.service_interval)

    policy_job.service_rate = [token.random_state.gamma(
         average_processing_time ** 2 / self.worker_variability,
                                              self.worker_variability /
                                                  average_processing_time) for
                            _ in range(self.number_of_users)]
```

Listing 3.2: User service rate sampling following an Erlang distribution

As previously mentioned, the Erlang distribution is a special case of the Gamma distribution where k defines the shape of the curve. This distribution is better suited to model service rates since with an appropriate k one can approximate a normal distribution without incurring in the aspect of having to manually reset negative values to one (thus loosing statistical generality). This is asserted by the formal definition of Erlang's support with $x \in [0, \infty)$.

`NumPy`'s implementation of its Erlang distribution is used[11]. Equation 3.2 defines the probability density function of the Erlang's distribution with the alternative parametrization that uses μ instead of λ as scale parameter, which is its reciprocal. This corresponds to the NumPy's implementation.

$$f(x; k, \mu) = \frac{x^{k-1} e^{-\frac{x}{\mu}}}{\mu^k (k-1)!} \text{ for } x, \mu \geq 0 \tag{3.2}$$

Each user task object has a claim token method, which takes tokens as input parameters and finally makes a call to its designed policy, passing the token. On this top level, without stepping into the single policies implementations, the logic is straightforward: start events generate tokens, user tasks that are direct children of start events claim the newly generated tokens, ask the designated policies to work the token assigned to them and finally, after a service interval timeout which corresponds to the user's specific service interval, they release the token. The logic can be seen in Listing 3.3.

```
def claim_token(self, token):
    token.worked_by(self)
    policy_job = self.policy.request(self, token)
    service_time = yield policy_job.request_event
    yield self.env.timeout(service_time)
    self.policy.release(policy_job)
```

Listing 3.3: User task claim method

Policy

Policies are a particular object that does not directly participate in the workflow processes, it merely serves a role as a general supervisor that has the whole overview of the process allowing it to operate on a more abstract level.

The implementation of the policy objects can be seen in Figure 3.2.

[11] https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.gamma.html (accessed: 06.01.2017)

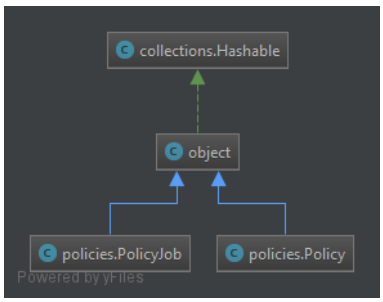

Figure 3.2: Policies abstract implementation

Each policy is a blueprint for the actual implementation of the policy itself. It holds minimal information such as a simulation environment, number of users and worker variability. The worker variability is a global parameter that is used for the service time generation as already explained in Subsection 3.1.3.

As a blueprint, each policy object defines two abstract method for requesting an optimal assignment for a specific token and for later releasing that token and effectively freeing the user that was busy working on it. Refer to Figure 3.3 for its implementation overview.

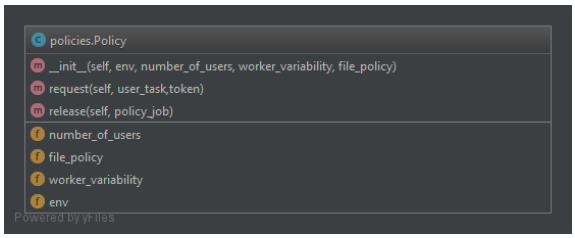

Figure 3.3: Policy methods and attributes

In its request method, each policy generates a policy job object, which is again an abstract implementation of a job that the policy will work in order to return an optimizes assignment to a user task. Each policy job requires a user task and a token object as initialization parameters in order to be uniquely identifiable inside the whole process. Moreover, each policy job object serves as a bookkeeping agent by storing and dumping useful information every time it status changes, such as arrival, assigned, started and finished times, assigned user and a list of service times for all available users. Refer to Figure 3.4 for its implementation overview.

3.2 Optimization Policies

Different types of policies have been implemented following the foundations laid by Zeng and Zhao as outlined in Subsection 2.1.2. In their work the authors investigate five "role-resolution" policies used for optimal task to user assignment [39, p. 7]. Following a brief description of the five aforementioned policies:

1. A load balancing policy consists in assigning a task as soon as it arrives to a qualified worker with the shortest task queue at that moment. In this policy workers execute tasks assigned

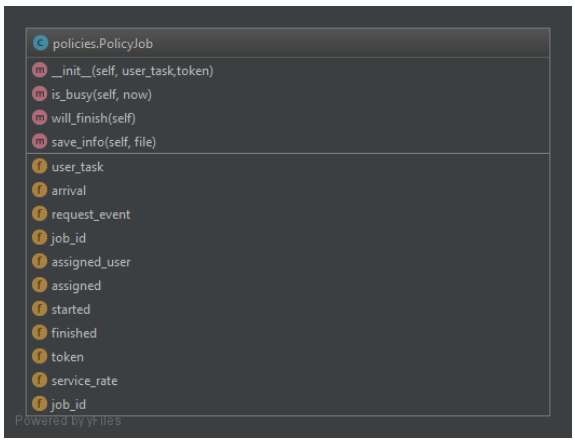

Figure 3.4: Policy job methods and attributes

to them on a FIFO fashion. The authors call this policy the "least loaded qualified person" or LLQP.

2. A policy that maintains a single queue being shared among all users is referred to the authors as "shared queue" or SQ policy.

3. Another policy that maintains both a shared queue among all users and each user having an own queue and transfers tasks from the former to the latter is called "K-Batch" policy. Transfer of tasks from the shared queue to users is done using an optimal task assignment procedure as soon as the shared queue reaches a critical batch size K.

4. The following policies takes the "K-Batch" policy but reduces the individual queue size of each user to one. This means that the optimization problem is still being solved as soon as the shared queue reaches the critical size K, however actual movement of tasks from the shared queue to the individual user queue happens only when user i is not busy *i.e.*, his individual queue is empty at simulation time t. This policy is called according to the authors as "K-Batch-1"

5. The last policy further simplifies the fourth by weakening the batch size constraint and reduces it to one. This means that the optimal task assignment procedure is executed immediately. This policy is referred by the authors as "1-Batch-1".

All batching policies require the solution of an optimization problem. The authors define this problem as "minimizing the maximum flowtime given the dynamic availability of the workers" and call it "minimizing sequential assignment (MSA)" [39, p. 7]. The authors define the task flowtime as the elapsed simulation time between task generation and its completion [4, 39]. Formally MSA is formulated as follows:

$$\min_{z} \quad z \tag{3.3}$$

subject to:

$$\sum_{i \in W} x_{ij} = 1 \quad \forall j \in T \tag{3.4}$$

$$a_i + \sum_{j \in T} x_{ij} p_{ij} \leq z \quad \forall i \in W \tag{3.5}$$

$$x_{ij} \quad \text{or} \quad x_{ij} = 1 \quad \forall i \in W, \forall j \in T \tag{3.6}$$

All variables definition still hold without loss of generality as defined by the authors [39, pp. 5-7].

The class inheritance structure of the policies implementation can be seen in Figure 3.5.

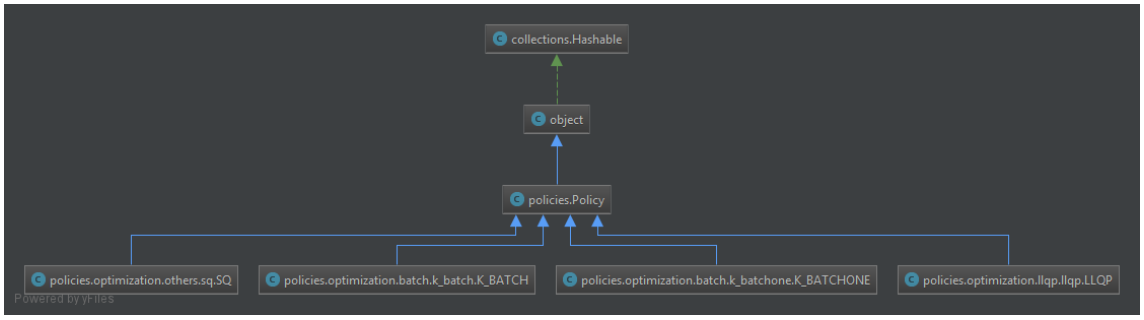

Figure 3.5: Policies class structure

The authors definition of the MSA problem is however a simplified version of the actual problem of "minimizing the maximum task flowtime" (MF) as defined by Baker [4] with consideration of the dynamic arrival of tasks problem, defined by the authors as the DMF problem [39]. The DMF problem is formally defined by Zeng as follows:

$$\min_{z} \quad z \tag{3.7}$$

subject to:

$$\sum_{i \in W} \sum_{k \in T} x_{ijk} = 1 \quad \forall j \in T \tag{3.8}$$

$$s_j \geq r_j \quad \forall j \in T \tag{3.9}$$

$$(x_{ijk} + x_{ij'(k+1)} - 1)(s_j + p_{ij}) \leq s_{j'} \quad \forall i \in W, \forall k \in T, \forall j \in T, \forall j' \in T \tag{3.10}$$

$$s_j + \sum_{i \in W} \sum_{k \in T} p_{ij} x_{ijk} - r_j \leq z \quad \forall j \in T \tag{3.11}$$

$$x_{ijk} = 0 \quad \text{or} \quad x_{ijk} = 1 \quad \forall i \in W, \forall j \in T, \forall k \in T \tag{3.12}$$

$$s_j \geq 0 \tag{3.13}$$

Again, all variables definition still hold without loss of generality as described by the authors [39, p. 6]. As Zeng notes in his work, Equation 3.10 contains nonlinear constraints but mentions that by adding auxiliary variables the aforementioned DMF formulation can be effectively converted into a mixed integer program and thus solve it [39, p. 6]. On this note Zeng

argues that the application of the DMF problem in practice poses some problems [39]. In this thesis however a conversion of the DMF formulation proposed by Zeng is formulated in order to adequately solve the optimization problem. The formal definition of such optimization problem is called EDMF (which stands for extended DMF) and is devised as follows:

$$\min_{z_{\max}} \quad z_{\max} \tag{3.14}$$

subject to:

$$\sum_{i \in W} \sum_{k \in T} x_{ijk} = 1 \quad \forall j \in T \tag{3.15}$$

$$a_i + \sum_{j \in T} p_{ij} x_{ijk} \leq z_{i*k} \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k = 0 \tag{3.16}$$

$$z_{i*k-1} + \sum_{j \in T} p_{ij} x_{ijk} \leq z_{i*k} \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k > 0 \tag{3.17}$$

$$z_{i*k} + \sum_{j \in T} w_j x_{ijk} \leq z_{\max} \quad \forall i \in W, \forall k \in T \tag{3.18}$$

$$\sum_{j \in T} x_{ijk} \leq 1 \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k = 0 \tag{3.19}$$

$$\sum_{j \in T} x_{ijk} \leq \sum_{j \in T} x_{ijk-1} \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k > 0 \tag{3.20}$$

$$z_{i*k} \geq 0 \quad \forall i \in W, \forall k \in T \tag{3.21}$$

This formulation clearly gets rid of the nonlinear constraints while still accounting for dynamical arrival of tasks, making thus the DMF problem as defined by Zeng effectively solvable.

When considering the minimization of the maximum flowtime of a task inside a process, the EDMF formulation can be further simplified by adopting some assumptions about the order and sequence of tasks. Based on how the batching policies are implemented, the policy job objects to be worked by users are implicitly stored in a sorted fashion. This means that the z helper variables defined for EDMF are not strictly necessary and thus can be compressed by Equation 3.22:

$$a_i + \sum_{t=1}^{k} \sum_{j} (p_{ij} + w_j I(t = k)) x_{ijt} \tag{3.22}$$

The whole concept consists in the introduction of an identity variable I which is true only if task j is currently being assigned as the kth task to user i, meaning that for this specific case also the waiting time for task j has to be accounted for. For all other cases *i.e.*, $j < k$ the identity variable I will not hold thus effectively zeroing the w_j variable.

Figure 3.6 depicts the potential scenario where three tasks are assigned to a specific user i following a sequence where task 2, 3 and j are assigned respectively as first, second and third tasks (thus respecting the k notation outlined above them).

In order to calculate z_{ijk}, one has to consider also when user i will actually be available to process his first task. This is depicted by the variable a_i, which summed together with the respective service times of user i for task j gives the complete work time user i will require to process all tasks assigned to him.

Without further ado, the simplified formulation of the extended DMF variant (called SDMF, which stands for simplified DMF) is the following:

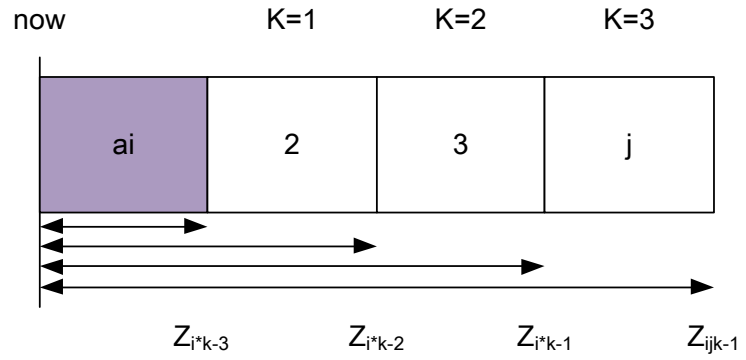

Figure 3.6: EDMF Task Assignment

$$\min_{z_{\max}} \quad z_{\max} \tag{3.23}$$

subject to:

$$\sum_{i \in W} \sum_{k \in T} x_{ijk} = 1 \quad \forall j \in T \tag{3.24}$$

$$a_i + \sum_{t=1}^{k} \sum_{j} (p_{ij} + w_j I(t = k)) x_{ijt} \leq z_{\max} \tag{3.25}$$

$$\sum_{j \in T} x_{ijk} \leq 1 \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k = 0 \tag{3.26}$$

$$\sum_{j \in T} x_{ijk} \leq \sum_{j \in T} x_{ijk-1} \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k > 0 \tag{3.27}$$

By comparing both formulation it is clear that SDMF manages to simplify the mathematical formulation and relaxing the required amount of constraints while still attaining the same level of effectiveness. Please note, however, that this simplification is only possible because of the nature of the implementation.

Based on this approach and by further exploiting the implicit order implementation of task arrival in the global queues for both batching policies, it is possible to argue that the k sequence indexing can be relaxed as well, thus even further simplifying the mathematical formulation and respectively the optimization problem size and computation costs.

The formulation of the DMF problem by relaxing both the z variables and k indexes, it is possible to formulate the same DMF problem as follows:

$$\min_{z_{\max}} \quad z_{\max} \tag{3.28}$$

subject to:

$$\sum_{i \in W} x_{ij} = 1 \quad \forall j \in T \tag{3.29}$$

$$a_i + \sum_{k=1}^{j} (p_{ik} + w_k I(k = j)) x_{ik} \leq z_{\max} \tag{3.30}$$

This formulation is colloquially called the ESDMF method (extremely simplified DMF).

This extremely simplified version is however only possible by the nature of its implementation. Since the both the global as well as the local queues are implemented as FIFO queues, it is possible to relax the ordering constraint from the mathematical formulation since it is already implicitly defined by the implementation.

Yet one last optimization of the method proposed by Zeng done in this thesis is a method that aims to optimally solve the assignment problem by changing its goal: minimize the service times by setting an upper bound on the maximum flowtime. This method uses a two step process in order to optimally solve the assignment problem: 1. optimally solve by means of using on of the DMF optimization methods previously outlined. This yields an upper bound for the maximum flowtime. 2. use this upper bound as a constraint for the actual optimization in order to effectively optimize the problem for the minimal service time amongst users, jobs and their corresponding service time.

$$\min_{z} \quad \sum_{i \in W} \sum_{k \in T} z_{ik} \tag{3.31}$$

subject to:

$$\sum_{i \in W} \sum_{k \in T} x_{ijk} = 1 \quad \forall j \in T \tag{3.32}$$

$$a_i + \sum_{j \in T} p_{ij} x_{ijk} - M(1 - \sum_{j \in T} x_{ijk}) \leq z_{i*k} \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k = 0 \tag{3.33}$$

$$z_{i*k-1} + \sum_{j \in T} p_{ij} x_{ijk} - M(1 - \sum_{j \in T} x_{ijk}) \leq z_{i*k} \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k > 0 \tag{3.34}$$

$$z_{i*k} + \sum_{j \in T} w_j x_{ijk} \leq z_{\max} + \epsilon \quad \forall i \in W, \forall k \in T \tag{3.35}$$

$$\sum_{j \in T} x_{ijk} \leq 1 \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k = 0 \tag{3.36}$$

$$\sum_{j \in T} x_{ijk} \leq \sum_{j \in T} x_{ijk-1} \quad \forall i \in W, \forall k \in T \quad \text{for} \quad k > 0 \tag{3.37}$$

$$z_{i*k} \geq 0 \quad \forall i \in W, \forall k \in T \tag{3.38}$$

$$M = \max_{a} a_i + \max_{p} \sum_{i \in W} \sum_{j \in T} p_{ij} \tag{3.39}$$

$$\epsilon = 1 \times 10^{-4} \tag{3.40}$$

Table 3.1 shows the computational complexity for the methods outlined in this chapter. The MSA method is the simplest solver and exhibits a linear complexity compared to the DMF method proposed by Zeng. As it can be seen the methods implemented in this thesis, specifically the

ESDMF method, both solves the DMF method and does it by keeping the same linear complexity as the MSA method. The ST method proposed in this thesis exhibits a higher computational complexity but achieves a better optimization. This trade-off however requires a more in depth explanation which will follow in Section 4.2.

Solver	Computation Costs
MSA	$O(mn)$
EDMF	$O(mn^2)$
SDMF	$O(mn^2)$
ESDMF	$O(mn)$
ST	$O(m^2n^2)$

Table 3.1: Comparison of computational costs for different solvers

3.3 Reinforcement Learning Theory

In this section the reinforcement learning approach used to solve the different role resolution problems is depicted. Initially a foundation basis in the required knowledge is depicted and afterwards the description of the analysis environment implementation is presented.

3.3.1 Reinforcement Learning Definition

Reinforcement learning is a novel approach originated as a branch from the broader field of machine learning. It is an automated approach to understanding and automating learning and decision-making [34, p. 15]. It distinguishes itself from other approaches by its novel focus on learning thanks to an agent which directly interacts with its environment, without the necessity of relying on training sets [34, p. 15].

The formal framework used by reinforcement learning defines the interaction between the so called learning agent and its environment by means of states, actions and rewards [34, p. 15].

Key concepts in the field of reinforcement learning are those of values and value functions which helps distinguish reinforcement learning methods from evolutionary methods which have to undergo scalar evaluations of entire policies [34, p. 15].

3.3.2 Finite Markov Decision Processes

Reinforcement learning approaches learn by interacting with the environment in order to achieve a goal. The agent interacting with the environment does this in a sequence of discrete time steps, it performs actions (choices made by the agent), reaches then states (basis for making decisions) and eventually receives rewards (basis for evaluating the choices) [34, p. 73]. Moreover, a policy is a stochastic rule that the agent relies upon to choose actions as a function of states [34, p. 73]. Ultimately, the sole goal of the agent is to maximize the reward that it receives over time [34, p. 73].

Returns are modeled as functions of future rewards that an agents must maximize [34, p. 73]. There exist two types of return functions which depend on the nature of the tasks and a discounting preference: 1. for episodic tasks a non discontinued approach is preferred while 2. for continuous tasks, however, a discounted approach is better suited [34, p. 73].

Equation 3.41 defines the sum of the rewards received over time step t:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots R_T \tag{3.41}$$

If we account for discounting, Equation 3.41 has to be slightly adapted by introducing a discounting factor γ and can be found in Equation 3.42:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.42}$$

where $0 \leq \gamma \leq 1$.

An environment with which one agent interacts, can satisfy a Markov property if the information contained at present effectively summarizes the past without affecting the capability of effectively predicting the future [34, p. 73]. If the Markov property is satisfied, then this environment is called a Markov decision process (MDP) [34, p. 73].

Last but not least, value functions are used to assign each state or state-action pair an expected return based on the policy used by the agent [34, p. 74]. Optimal value functions assign the highest achievable return by any policy to a state or state-action pair and such policies, whose values are optimal, are called optimal policies [34, p. 74].

Optimal state-value functions v_* are formally defined as follows:

$$v_*(s) \doteq \max_{\pi} v_\pi(s) \tag{3.43}$$

whereas optimal action-value functions q_* are formally defined as follows:

$$q_*(s, a) \doteq \max_{\pi} q_\pi(s, a) \tag{3.44}$$

3.3.3 Dynamic Programming

Dynamic programming (DP) is a set of ideas and algorithms that can be used to solve MDPs [34, p. 95]. There are two approaches in dynamic programming for solving MDPs: 1. policy evaluations is the iterative computation of value functions of a given policy and 2. policy improvement is the idea of computing an improved policy under the conditions of its given value functions [34, p. 95].

By combining these two approaches we obtain the two most notable DP methods *i.e.*, policy and value iteration [34, p. 95].

One captivating property of DP methods is the concept of bootstrapping: updating estimates of values of states by approximating the values of future states [34, p. 96].

3.3.4 Monte Carlo Methods

Monte Carlo (MC) methods use experience in form of sample episodes in order to learn value functions and optimal policies [34, p. 123]. This approach yields different advantages over the DP methods seen in Subsection 3.3.3: 1. they do not need a model of the environment's dynamics as they learn the optimal solutions by merely interacting with the environment, 2. since the learn from sample episodes, they are very well suited for simulation environments, 3. it is efficient and surprisingly easy to use MC methods to focus on smaller regions or subsets of a problem and 4. MC methods are more robust when it comes to violations of the Markov property since they do not bootstrap for updating their values [34, p. 123].

One of the drawbacks that MC methods bring along is the concept of maintaining sufficient exploration: by always acting greedily, alternative states will never yield their returns thus potentially never learning that they might prove to be better [34, p. 123].

A MC simplified method can be formally defined as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{3.45}$$

where G_t is the discounted return function defined by Equation 3.42 and α is a constant step-size parameter [34, p. 127]. MC methods must wait until the end of one episode in order to evaluate the incremental value of $V(S_t)$ since only at that point in time G_t is known [34, p. 128].

3.3.5 Temporal-Difference Learning

Temporal-difference (TD) are yet another set of learning methods for reinforcement learning. Compared to the MC methods explained in Subsection 3.3.4, TD methods do not need to wait all the way up to the end of an episode to actually learn, they only must wait until the next step *i.e.*, they can bootstrap [34, p. 128]. When they reach time step $t+1$, they observe a reward R_{t+1} which then use to estimate $V(S_{t+1})$ [34, p. 128]. The simplest TD method, which is called $TD(0)$, is defined as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{3.46}$$

TD methods, same as MC methods, are not excluded from the sufficient exploration methods [34, p. 147]. TD methods deal with this complication in two different ways: 1. on policy by using an algorithm called Sarsa and 2. off policy by using an algorithm called Q-learning [34, p. 128].

3.3.6 On-policy Prediction with Approximation

Up until now, the different methods presented are not suited for arbitrarily large state spaces. However, there exist solution to tackle such large state spaces: approximate solution methods. Under the assumption that one must always account for finite and limited computational resources, it is not feasible to find an optimal policy or value function, instead we have to settle for a good approximation of the solution [34, p. 189].

An essential characteristic for reinforcement learning algorithms venturing in the area of approximation is being able of generalization *i.e.*, using experience from a limited subset of the state space to effectively generalize and produce a valid approximation of a much larger subset [34, p. 189]. Reinforcement learning methods are capable of achieving this by relying on supervised-learning function approximation which essentially use backups as training example [34, p. 222]. Specifically, one brilliant set of methods are those using parametrized function approximation *i.e.*, the policy is parametrized by a weight vector θ.

The parametrized functional form with weight vector θ can be used to write $\hat{v}(s,\theta) \approx v_\pi(s)$, which is the approximated value of state s given weight vector θ [34, p. 191].

It is then clear that the weight vector θ has to be chosen wisely: this can be done by using variations of stochastic gradient descent (SGD) methods [34, p. 223]. SGD methods adjust the weight vector after each step by a tiny amount following the direction that would reduce the error the most:

$$\theta_{t+1} \doteq \theta_t - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t,\theta_t)]^2 \tag{3.47}$$

$$= \theta_t + \alpha[v_\pi(S_t) - \hat{v}(S_t,\theta_t)]\nabla\hat{v}(S_t,\theta_t) \tag{3.48}$$

where α is a positive step size parameter and $\nabla f(\theta)$ is the vector of partial derivatives with respect to θ:

$$\nabla f(\theta) \doteq \left(\frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, \ldots, \frac{\partial f(\theta)}{\partial \theta_n} \right)^\top \tag{3.49}$$

CITE ▷*add book citation*◁

An exceptional case is linear methods for function approximation, where the approximate function $\hat{v}(\cdot, \theta)$ is a linear function of the weight vector θ [34, p. 198]. This means that for each state s there is a corresponding vector of features $\phi(s) \doteq (\phi_1(s), \phi_2(s), \ldots, \phi_n(s))^\top$ which has the same number of components as θ [34, p. 198]. With this definition in mind, we can now formally define the state-value function approximation as the inner product between θ and $\phi(s)$:

$$\hat{v}(s, \theta) \doteq \theta^\top \phi(s) \doteq \sum_{i=1}^{n} \theta_i \phi_i(s) \tag{3.50}$$

This simplified case of linear function approximation for state-value functions finally brings us to how we can use the SGD:

$$\nabla \hat{v}(s, \theta) = \phi(s) \tag{3.51}$$

Equation 3.51 tells us that for the simple linear case the SGD is nothing more than the corresponding features value [34, p. 199].

Artificial Neural Networks

Up to this point, we considered linear function approximation. Artificial neural networks (ANNs) can be used instead for nonlinear function approximation [34, p. 199]. The simplest case of an ANN is a single feedforward perceptron, meaning that it has only one hidden layer (*i.e.*, a layer that is neither an input nor an output layer) and that the ANN at hand has no loops between its neurons, meaning that the output can not influence the input (compared to recurrent neural networks, in which the output indeed can influence the input) [34, p. 216].

The connections between neurons inside an ANN are called weights and the analogy to its human counterpart is how strong synaptic connections are [34, p. 216]. Refer to Figure 3.7 for a depiction of a single layer ANN.

The key about solving nonlinearity with ANN is how they apply nonlinear functions to the sum of their weights, this process is done by means of activation functions [34, p. 216]. There are different types of activation functions that can be used but each one of the usually exhibits an "S" shape (*i.e.*, sigmoid), such as the sigmoid $\sigma(x) = \frac{1}{(1+e^{-x})}$, the $\tanh(x) = 2\sigma(2x) - 1$ or the different classes of rectified linear unit (ReLU), which have become extremely popular in the last few years due to their peculiar characteristics such as $f(x) = \max(0, x)$ [34, p. 216].

Even though single layer perceptrons are powerful enough to approximate nonlinearity, in the past years a development toward more complex ANN with multiple hidden layers (*i.e.*, multilayer perceptrons) has been on the rise [34, p. 217]. These complex ANN allow to solve many artificial intelligence tasks in a much more efficient way [5]. This area is called deep reinforcement learning and it has shed light on solutions that were never though possible before (for practical applications refer to Mnih's [23] and Silver's work [28])[12] [5]. Refer to Figure 3.8 for a depiction of a multi layer ANN.

[12] Note that the class of deep ANN used in these works is a particular one called deep convolutional networks, which are specialized networks used for processing high dimensional data arranged in spatial arrays like images [34, p. 219], [19].

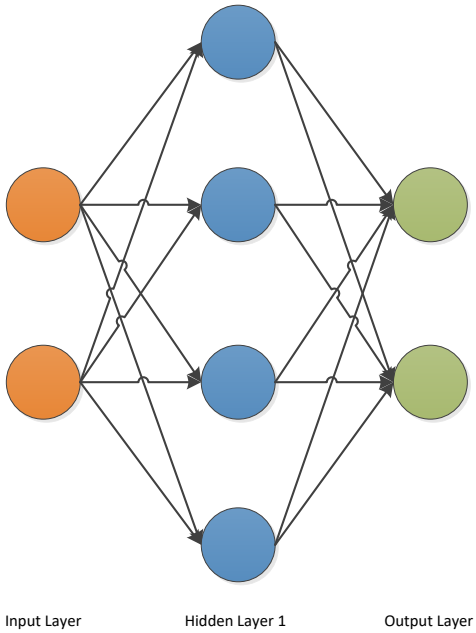

Figure 3.7: Single layer ANN

Despite appearing more complex, ANN rely on a similar approach for learning (*i.e.*, updating their internal parameters, or in this case the whole network synaptic connections) based on the SGD method outlined in Subsection 3.3.6 [34, p. 217]. This algorithm is called backpropagation and consists of doing a forward pass in which the activation function of each neuron is computed and then a backward pass computes the partial derivatives for each synaptic connection [34, p. 218].

As any other function approximation method, overfitting can be a problem for ANN as well and it is particularly present for deep ANN [34, p. 218]. There are different techniques that can be used in order to mitigate this effect, with the most prominent one being the dropout method outlined by Srivastava [31].

3.3.7 On-policy Control with Approximation

Moving towards control with value function approximation, we now focus on the approximation of the action-value function $\hat{q}(s, a, \theta) \approx q_*(s, a)$ [34, p. 229].

For the special case of the so called one-step Sarsa method, its gradient-descent update for the action-value function is defined as follows:

$$\theta_{t+1} \doteq \theta_t + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \theta_t) - \hat{q}(S_t, A_t, \theta_t)]\nabla\hat{q}(S_t, A_t, \theta_t) \tag{3.52}$$

and this method has excellent good convergence properties towards optimality [34, p. 230].

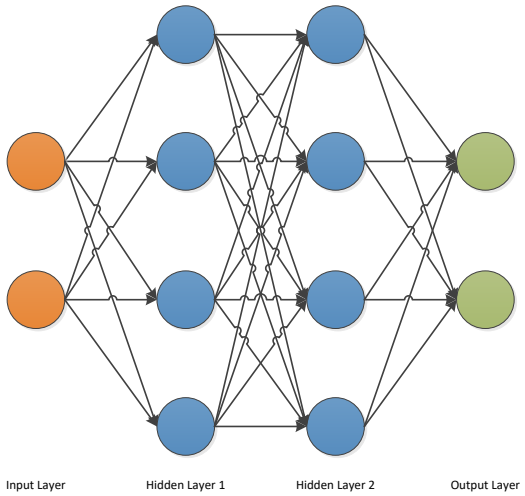

Figure 3.8: Multi layer ANN

3.3.8 Off-policy Methods with Approximation

When moving towards the field of off-policy learning, one of the biggest problems that one might incur in is the convergence problem: off-policy learning with approximation is considerably harder compared to its tabular counterpart [34, p. 243]. Off-policy learning defines two policies, π and μ, where the former is the value function we seek to learn based on the latter [34, p. 243].

A new aspect being introduced in off-policy learning is the importance sampling concept, formally defined as follow:

$$\rho_t \doteq \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \tag{3.53}$$

which can be used to "warp the update distribution back to the on-policy distribution, so that semi-gradient methods are guaranteed to converge." [34, p. 243].

During off-policy learning, both policies π and μ are usually defined as full greedy and somewhat more exploratory ϵ-greedy [34, p. 243].

For the purpose of this thesis, the focus has been put upon the episodic action value update algorithms, defined as follows:

$$\theta_{t+1} \doteq \theta_t + \alpha\delta_t \nabla\hat{q}(S_t, A_t, \theta_t) \tag{3.54}$$

$$\delta_t \doteq R_{t+1} + \gamma \underbrace{\sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \theta_t)}_{\max_a \hat{q}(S_{t+1}, a, \theta_t)} - \hat{q}(S_t, A_t, \theta_t) \tag{3.55}$$

what is important to note here, is that the episodic off-policy algorithm does not use importance sampling as defined by Equation 3.53 [34, p. 244]. The authors state that this approach is clear for tabular methods but it is rather a "judgment call" for methods using approximation functions and deeper understanding of the theory of function approximation is needed [34, p. 244].

3.3.9 Policy Gradient Methods

Up until now all methods were based on the concept of learning values of actions and subsequently choosing the correct actions based on estimates, however, we now move our focus towards methods that actually learn a parametrized policy without needing value functions at all[13] [34, p. 265]. Parametrized policies work with probabilities that a specific action a will be chosen at time t if the agent finds itself in state s at time t with a weight vector θ [34, p. 265]. For policy gradient methods it is crucial to learn the weight vector based on a performance measure $\eta(\theta)$ by trying to maximize and thus approximating the gradient ascent of η as follows:

$$\theta_{t+1} \doteq \theta_t + \alpha\widehat{\nabla\eta(\theta_t)} \tag{3.56}$$

where $\widehat{\nabla\eta(\theta_t)}$ is nothing else than a stochastic estimate that approximates the gradient of $\eta(\theta)$ [34, p. 265].

For discrete action spaces, a suitable solution consists in forming parametrized numerical preferences $h(s, a, \theta) \in \mathbb{R}$ [34, p. 266]. This means that the best actions is given the highest probability according to a softmax distribution:

$$\pi(a|s, \theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}} \tag{3.57}$$

where $e \approx 2.71828$ [34, p. 266].

Moreover, the preferences can be, as previously mentioned:

$$h(s, a, \theta) \doteq \theta^\top \phi(s, a) \tag{3.58}$$

i.e., simply linear in features [34, p. 266].

With these definitions in mind, one can formally define one of the very first Monte Carlo based policy gradient methods: REINFORCE [38].

Williams defines his REINFORCE algorithm by the following update function:

$$\theta_{t+1} \doteq \theta_t + \alpha\gamma^t G_t \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \tag{3.59}$$

note that the vector $\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$ is called eligibility vector and it is usually written in a more compact form of $\nabla \log \pi(A_t|S_t, \theta)$ by relying on the mathematical identity $\nabla \log x = \frac{\nabla x}{x}$ [34, p. 271].

For the REINFORCE algorithm, its eligibility vector is defined as follows:

$$\nabla_\theta \log \pi(a|s, \theta) = \phi(s, a) - \sum_b \pi(b|s, \theta)\phi(s, b) \tag{3.60}$$

and this method has solid convergence properties [34, p. 271].

Policy Gradient with Baseline

REINFORCE, however, being a method based on Monte Carlo it might exhibit high variance and prove relatively slow in its learning rate [34, p. 271]. By introducing a baseline $b(s)$ to compare the action value:

[13] Actor-critic methods are an exception, where a learned value function is used in combination with policy gradient as a baseline in order to lower variance.

$$\theta_{t+1} \doteq \theta_t + \alpha(G_t - \overbrace{b(S_t)}^{\text{baseline}})\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{3.61}$$

one can achieve a positive effect towards diminishing variance of the update rule [34, p. 271].

Actor-Critic Policy Gradient

By introducing a base line, we have seen that variance can be lowered, however, the REINFORCE algorithm with a baseline is not a proper actor-critic method as its state-value function is only used as baseline and not as a critic *i.e.*, it is not used for bootstrapping [34, p. 273]. By introducing bootstrapping we introduce bias and dependence of the quality of the approximated function, which in turn help to reduce variance and learn faster [34, p. 273].

The only negative aspect still remaining is that policy gradient methods are still based on a full Monte Carlo update trajectory: this can be also mitigated by replacing the update function by temporal difference learning approaches, such as those defined in Subsection 3.3.5 [34, p. 273].

The formal definition of a one-step actor-critic update method is depicted as follows:

$$\theta_{t+1} \doteq \theta_t + \alpha(R_{t+1} + \overbrace{\gamma\hat{v}(S_{t+1},w) - \hat{v}(S_t,w)}^{\text{TD update}})\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{3.62}$$

and this is now a fully online implementation that executes updated after each newly visited state [34, p. 274].

3.4 Reinforcement Learning Policies

Analog to Section 3.2, the same policies are considered where now different reinforcement learning methods and techniques are used to solve the assignment problem.

Different subsections are used in order to separate better the different approaches used for each type of policy: 1. Batch policy methods. 2. LLQP policy methods. 3. other policy methods that do not fit in any of the previous categories.

The two key concepts required in order to effectively apply reinforcement learning techniques in the domain of workflow processes and the optimal assignment of jobs to users are:

1. correctly defined states and actions spaces.

2. precise rewards definition.

3. effective update method for the policy's internal parameters.

In the next subsections a distinction between prediction and update methods is outlined.

3.4.1 Prediction and Control Methods

As previously outlined in Subsection 3.3.6, Subsection 3.3.7, Subsection 3.3.8 and Subsection 3.3.9 there are different prediction and control methods that can be applied.

Value Function Approximation

As mentioned in Section 3.4, it is crucial to correctly define the states and actions space for each problem. Each request that the policy receives generates a policy job object which is then passed in the internal evaluate method of the policy. Inside this method, for each policy job the state space S is defined as a $n \times n+1$ matrix which contains all busy times of the potential candidates (*i.e.*, users) concatenated to the user's current service time for the job. Formally the state space is defined as depicted in Equation 3.63:

$$S_{n,n+1} = \begin{bmatrix} a_1 & \cdots & a_1 \\ a_2 & \cdots & a_2 \\ \vdots & \ddots & \vdots \\ p_{1,j} & \cdots & p_{i,j} \end{bmatrix} \tag{3.63}$$

Since the possible actions are represented by the number of users, the state space is modeled such that for each possible actions a 1-D vector containing all busy times plus the service time of the user are present.

During the evaluation phase of a job the policy has to choose an action (*i.e.*, a user) by taking into account the current state space and its internal θ parameters. By using a state-value function approximation as defined in Equation 3.50, the policy evaluates the highest score for each possible user.

As an example, let us consider a snippet of how a K-Batch policy using a linear state-value function approximation performs its choices during its greedy phase: it iterates over all possible actions and performs the dot product between the state space and the corresponding θ vector and then maximizes the returned Q value. This approach can be seen in Listing 3.4

```
if self.greedy:
    action = max(range(self.number_of_users), key=lambda action: self.q(
        state_space, action))
else:
    rnd = self.EPSILON_GREEDY_RANDOM_STATE.rand()
    if rnd < self.epsilon:
        action = self.EPSILON_GREEDY_RANDOM_STATE.randint(0, self.
            number_of_users)
    else:
        action = max(range(self.number_of_users), key=lambda action: self.q(
            state_space, action))
```

Listing 3.4: Epsilon greedy approach

and its respective state-value function approximation in Listing 3.5.

```
def q(self, states, action):
    features = self.features(states, action)
    q = np.dot(features[action], self.theta[action])
    return q
```

Listing 3.5: State-value function approximation

Q values are however only one part of the requirements set by reinforcement learning methods, the next crucial aspect is defining the reward function. Since reinforcement learning agents are able to back-propagate what they have learned from one episode and thus update their internal factors, correctly defining a reward is a must. Since the goal for our domain is minimizing the

maximum flowtime (from now on this metric will be referred to as lateness) of a job, the reward itself corresponds to the lateness of a job during a specific task. This can be evaluated a priori since for each policy job we know its internal parameters required to calculate the lateness *i.e.*, busy time of user i plus the service time of user i for job j, or formally $a_i + p_{ij}$.

The last definition required in order to effectively apply the update on the policy's internal parameters θ is defining the SGD method as outlined by Equation 3.51. This method will give us the direction in which we have to update our internal θ parameters during our chosen update method and it is nothing more than the features themselves. As an example, refer to Listing 3.6 for the concrete implementation.

```
def features(self, states, action):
    features = np.zeros((self.number_of_users, self.number_of_users + 1))
    features[action] = states[action]
    return features
```

Listing 3.6: Features definition

The features method outputs a matrix which has its values populated only for the actual chosen action. Let us assume our policy has chosen user 1 out of 2 possible users, then the state space looks as defined by Equation 3.64:

$$S_{2,3} = \begin{bmatrix} a_1 & a_1 \\ a_2 & a_2 \\ p_{1,j} & p_{2,j} \end{bmatrix} \tag{3.64}$$

and its features vector looks as defined by Equation 3.65:

$$F_{2,3} = \begin{bmatrix} a_1 & 0 \\ a_2 & 0 \\ p_{1,j} & 0 \end{bmatrix} \tag{3.65}$$

Policy Gradient

With policy gradient methods the approach on how an action is chosen is shifted. Instead of maximizing a Q value through internal θ parameters in order to choose the "best greedy" action, we now have probabilistic choices. As already outlined in Subsection 3.3.9, having a probabilistic policy π means that the best action is now chosen according to the highest probability which follows a softmax distribution as defined in Equation 3.57 and its implementation can be seen in Listing 3.7.

```
def policy_probabilities(self, busy_times):
    probabilities = [None] * self.number_of_users
    for action in range(self.number_of_users):
        probabilities[action] = np.exp(np.dot(self.features(busy_times, action
            ), self.theta)) / sum(
            np.exp(np.dot(self.features(busy_times, a), self.theta)) for a in
                range(self.number_of_users))
    return probabilities
```

Listing 3.7: Softmax distribution of preferences probabilities

The policy probabilities method takes as input parameter the current state space and computes for each user its probability according to the current internal θ parameter as defined in Equation 3.58. The result of this method is a probabilities (or weights) 1-D vector corresponding

to a preference to assign a job to a specific user, where the index of the vector corresponds to the user and the value to its preference. Based on this preferences vector, the policy then computes a weighted random choice among all users, as can be seen in Listing 3.8.

```
chosen_action = self.RANDOM_STATE_PROBABILITIES.choice(self.number_of_users,
    p=probabilities)
```

Listing 3.8: Probabilistic user choice

Neural Networks as Function Approximation

Up to this point we have used linear functions for the approximation of the Q value for the different policies. As mentioned in Subsection 3.3.6, ANN can be used for nonlinear function approximation. The assignment problem poses its very well for this kind of application, in which we model our input layer as a 1-D vector containing all required information such as waiting time w of job j, service time p_{ij} of user i for job j and busy time a_i of user i. By following a policy gradient approach, we can categorize the output layer of our ANN using a softmax categorization function, mapping the preferences of job j to user i assignment as probabilities. Listing 3.9 show the modeling of a single layer perceptron in Tensorflow: one hidden layer connects the state space (*i.e.*, input to the ANN) together with its weights and biases, creates an activation function and maps the prediction layer (*i.e.*, output) with a softmax classification.

```
with tf.name_scope("neural_network"):
    layer_1 = tf.add(tf.matmul(state_space_input, weights['h1']), biases['b1'
        ])
    layer_1 = tf.nn.elu(layer_1)
    pred = [tf.add(tf.matmul(layer_1, weights['out'][b]), biases['out'][b])
        for b in range(batch_input)]
    probabilities = [tf.nn.softmax(pred[b]) for b in range(batch_input)]
```

Listing 3.9: Modeling of a single perceptron in Tensorflow

In order to update the ANN, a backpropagation has to take place. Such an update can both be made following a MC or TD approach (refer to Subsection 3.4.2 for a detailed distinction between these two update methods). As outlined in Subsection 3.3.6, we follow a SGD approach in which we update all the synaptic connections by computing the partial derivatives of all the weights. Listing 3.10 shows how the backpropagation for an ANN following a MC update method is done.

```
def train(self):
    for t, (state, output, choices) in enumerate(self.history):
        disc_rewards = self.discount_rewards(t)
        tmp_choices = [choice for choice in choices if choice is not None]
        for job_index, chosen_user in enumerate(tmp_choices):
            prob_value = output[job_index].flatten()[chosen_user]
            reward = disc_rewards[job_index]
            factor = reward / prob_value
            grad_input = np.zeros((self.number_of_users, 1))
            grad_input[chosen_user] = 1.0
            self.sess.run(self.apply[job_index], {self.state_space_input: state
                , self.gradient_input: grad_input, self.factor_input: factor})
```

Listing 3.10: Backpropagation algorithm following a MC update approach

3.4.2 Update Methods

As outlined in Subsection 3.3.4 and Subsection 3.3.5, there are mainly two different methods to update the policy's internal θ parameters *i.e.*, MC and TD. Let us take the example outlined by Sutton and Barto of leaving the office and getting home and the respective updates proposed by the two update methods [34, p. 130]. Figure 3.9 shows the graphical updates proposed by the two update methods.

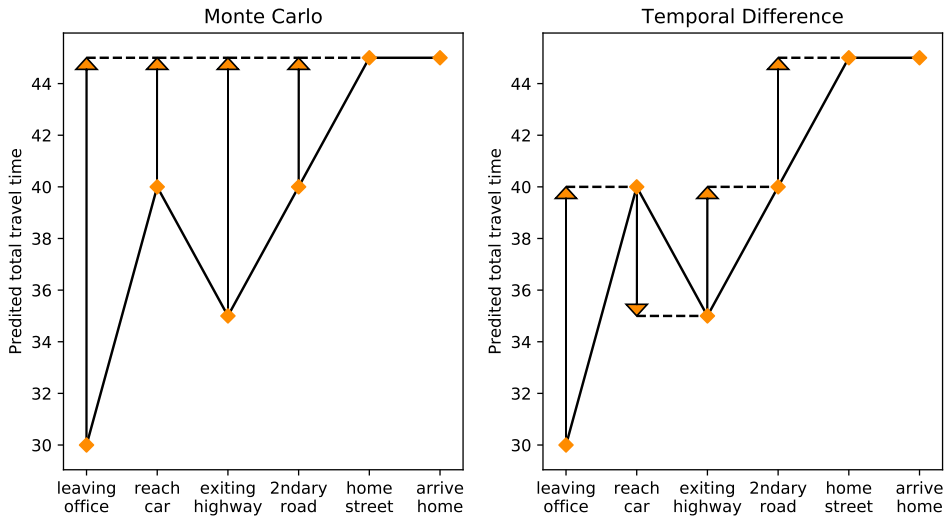

Figure 3.9: MC and TD proposed updates comparison (own plot based on Sutton and Barto)

As it can be clearly seen, the main deference lays in when the actual updating takes place. On one hand, the MC method needs to reach the end of an entire episode (*i.e.*, here it consists of actually arriving home) in order to fully back-propagate its learned value and update the θ parameters. On the other hand, TD is much more flexible and robust since it executes its updates at each time step, hence its name: temporal difference. For a formal overview of the difference between the two update methods, refer to Equation 3.45 for the MC update and to Equation 3.46 for the TD update. For the case at hand, this means that training the policies has to be done in a different fashion for the two update method: while TD based policies can be updated "on-the-fly", MC methods require batch training sessions (*i.e.*, episodes) at the end of which they can effectively learn and update their internal θ parameters to be used for the next episode. Not only the training approach is different, but the logic of the policy itself is also different: for TD based policies, the update method is being called internally since the policy knows its temporal steps, while for MC based policies the policy itself can not know a priori when an episode will finish and thus must relay on an "artificial" definition of such. The overall overhead is also different since MC based policies have to keep track of their whole episode history which usually is composed of the state space, chosen action and reward at time step t.

3.5 Hypothesis

3.6 Data

Chapter 4

Empirical Analysis

4.1 Methodology

In order to consistently and fairly evaluate all policies with the methods defined in the previous chapters, the following methodology was put in place:

- Each policy has its own simulation script that initialized a process that uses the predefined policy as means to optimally assign jobs to tasks
- Parameters are centrally defined.
- Different KPIs have been defined which are used to assert the efficiency of one policy against one another.

4.1.1 Simulation script

Each simulation script is the abstract element that import all required dependencies, initializes the SimPy simulation environment, the statistics file into which the policy dumps all data on runtime, the policy object itself to be used for the assignment and the workflow process to be used.

```
import simpy
from evaluation.statistics import calculate_statistics
from evaluation.subplot_evolution import evolution
from policies.optimization.batch.k_batch import K_BATCH
from simulations import *
from solvers.dmf_solver import dmf

policy_name = "{}_BATCH_DMF_NU{}_GI{}_SIM{}".format(BATCH_SIZE,
    NUMBER_OF_USERS, GENERATION_INTERVAL, SEED, SIM_TIME)

env = simpy.Environment()

file_policy = create_files("{}.csv".format(policy_name))

policy = K_BATCH(env, NUMBER_OF_USERS, WORKER_VARIABILITY, file_policy,
    BATCH_SIZE, dmf)
```

```
start_event = acquisition_process(env,policy,1,GENERATION_INTERVAL,False,
    None,None,None)

env.process(start_event.generate_tokens())

env.run(until=SIM_TIME)

file_policy.close()

calculate_statistics(file_policy.name, outfile=True)

evolution(file_policy.name, outfile=True)
```

Listing 4.1: Example of the structure of a simulation script. Here for the K-Batch policy using the DMF solver

The script initialized the chosen workflow process and then calls the tokens generation method of the start event. Eventually the whole simulation is started by calling the run method of the SimPy environment.

4.1.2 Workflow Process Modeling

Two different types of processes have been defined: 1. Consisting of only one user task. 2. A complex workflow process that is modeled against an acquisition process used in the real estate field for the acquisition of real estate properties.

Figure 4.1 illustrates the simple process.

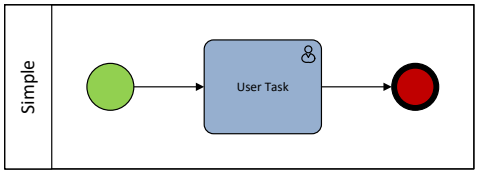

Figure 4.1: Simple workflow process consisting of only one user task

Figure 4.2 illustrates the complex acquisition process.

4.1.3 Central Simulation and Process Parameters Definition

One key aspect in order to assert fairness across policies while simulated is to centrally define all parameters. Listing 4.2 shows the key central parameters defined as global variables.

```
NUMBER_OF_USERS = 2
SERVICE_INTERVAL = 1
GENERATION_INTERVAL = 3
SIM_TIME = 100
BATCH_SIZE = 1
```

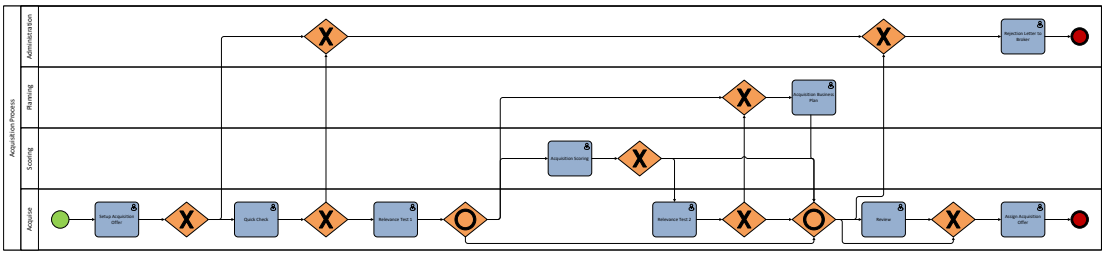

Figure 4.2: Acquisition workflow process consisting of multiple user tasks and decision nodes

```
TASK_VARIABILITY = 0.2 * SERVICE_INTERVAL
WORKER_VARIABILITY = 0.2 * SERVICE_INTERVAL
SEED = 2
```

Listing 4.2: Central parameters definition that ensures fairness across simulation runs

4.1.4 KPIs for Asserting Policy's Efficacy and Data Visualization

Based on Pinedo's and Zeng's definitions in their work, different KPIs have been defined to assert the efficacy of a policy, such as lateness, waiting time, service time, number of tokens completed, user loads and system load [24], [39]. Following the formal definitions of the per token j KPIs in respect to lateness L_j, wait time w_j, service time p_{ij} of assigned user i, arrival time A_j, assignment time a_j, start time S_j and finish time F_j.

$$L_j = F_j - A_j \tag{4.1}$$
$$w_j = S_j - A_j \tag{4.2}$$
$$p_{ij} = F_j - S_j \tag{4.3}$$

Moreover, if we account for simulation time T, load l_i of user i is defined as the sum of all service times of tokens that have been assigned to him during the simulation divided by the total simulation time T, or formally:

$$l_i = \frac{\sum_j p_{ij}}{T} \tag{4.4}$$

and thus the average system load $\bar{l}$ over all users participating n is defined as the average across all user's loads *i.e.*,

$$\bar{l} = \frac{\sum_i l_i}{n} \tag{4.5}$$

A summary plot with all KPIs is done for each simulation script. Figure 4.3 shows an example of how this summary looks like.

Additionally, for a more in depth visualization of a policy's performance, an evolution plot is also necessary. All types of policies share a common queues configuration, with a single global queue and a user specific queue. Each policy defines the maximal threshold a specific queue can reach. For a detailed explanation of the queues conformation refer to Section 3.2.

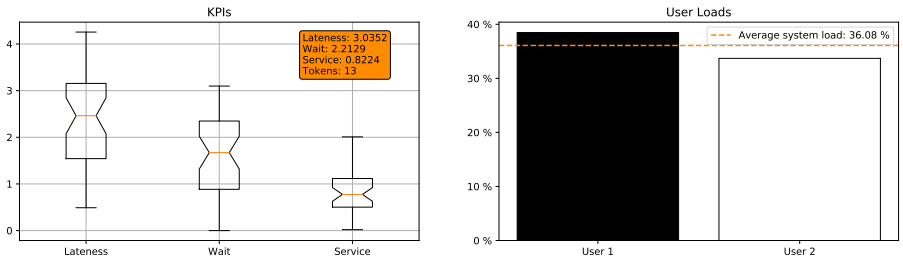

Figure 4.3: KPIs summary plot for a 3-Batch policy using the MSA solver, with two users, generation interval set to three and simulation time T set to 50

The evolution plot shows the state change for the policy being analyzed by plotting the flow of a token across different user tasks. Figure 4.4 shows such an example.

4.2 Optimization

This section focuses on the results of the different types of policies using the optimization solvers outlined in Section 3.2. All simulations have been tested with different combination of global variables *i.e.*, number of users, service interval, generation interval, length of simulation time, batch size (where it applies), task variability, worker variability and random state seed (where it applies). For ease of reading purposes, the global variables have been set to the following parameters according to the default column in Table 4.1.

Variable	Default	Valid Range
Number of Users	3	$1 - \infty$
Service Interval	1	$1 - \infty$
Generation Interval	3	$1 - \infty$
Simulation Time	50	$1 - \infty$
Batch Size	3	$1 - \infty$
Task Varaibility	20%	$0\% - 100\%$
Worker Variability	20%	$0\% - 100\%$
Random State Seed	2	$\emptyset - \infty$
Workflow Process	Acquisition	Acquisition, Simple

Table 4.1: Global Parameters for Simulation

4.2.1 Comparison with Existing Literature

Zeng outlines in his work how different global parameters configurations and policy usage can affect KPIs [39, pp. 18-22]. He summarizes his key findings as follows: 1. Usage of batch optimization should be done only with medium to high system load [39, p. 24]. 2. Batch optimization policies without a fixed batch size, such as 1-Batch-1 yield best results [39, p. 24].

In order to assert the validity of the interpretation of Zeng's works and all subsequent deriva-

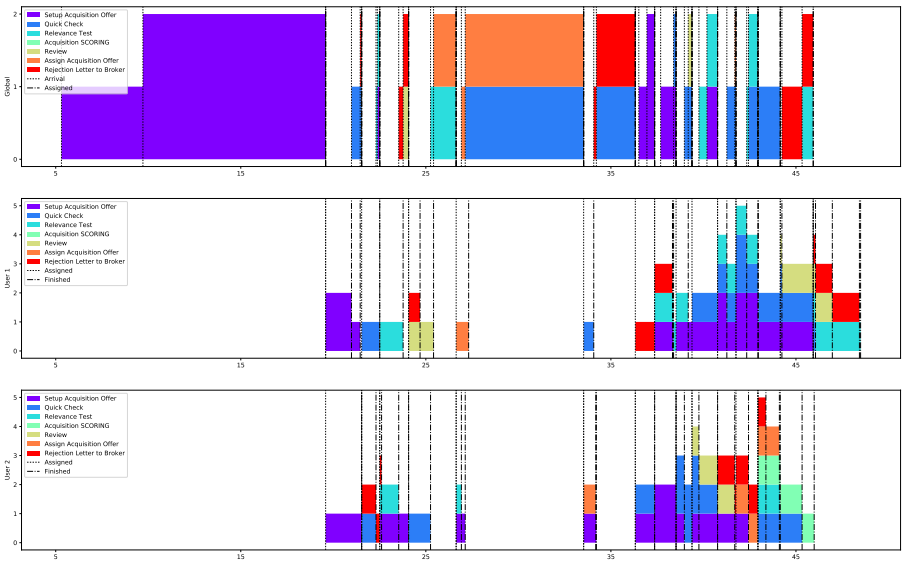

Figure 4.4: Evolution plot for a 3-Batch policy using the MSA solver, with two users, generation interval set to three and simulation time T set to 50

tive policies a comparison with similar configurations has been made for all five optimization policies. Zeng's main efficiency parameter is defined as the maximum flowtime or "In business terms, maximum flowtime represents the guaranteed response time across tasks, indicating the quality of services" [39, p. 17]. In this study, the comparable parameter used to evaluate a policy's efficiency is called lateness and has been previously defined in Equation 4.1.

In regards to lateness, Figure 4.5 shows that akin results are obtained.

The simulation have been run with the parameters outlined in tab by using the same solver used by Zeng: MSA.

By running the same simulations with the optimization solver implemented for this thesis (*i.e.*, ST, refer to Section 3.2), ceteris paribus, the summarized KPIs amongst all optimization policies can be seen in Figure 4.6.

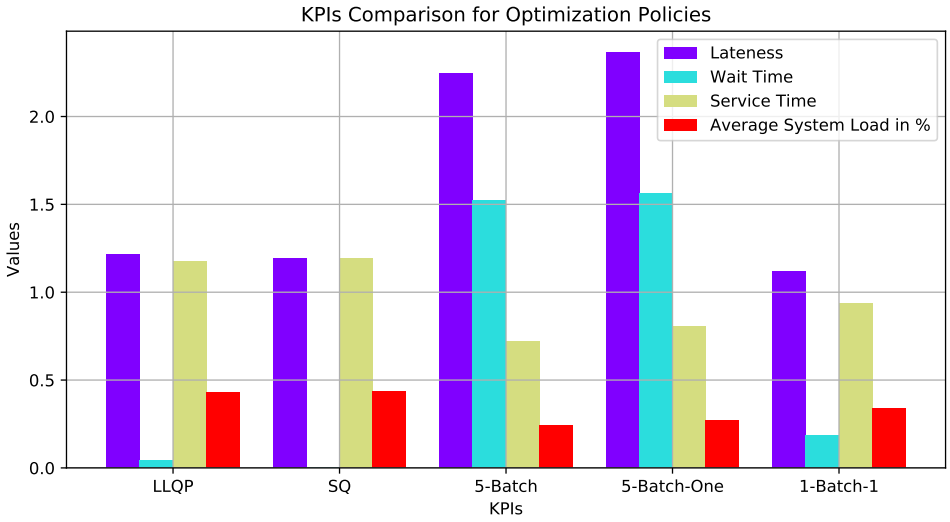

Figure 4.5: KPIs comparison for different optimization policies using the MSA solver

Variable	Value
Number of Users	5
Service Interval	1
Generation Interval	2
Simulation Time	50
Batch Size	5 (1 for 1-Batch-1)
Task Varaibility	20%
Worker Variability	20%
Random State Seed	2
Workflow Process	Acquisition

Table 4.2: Global Parameters for Optimization Policies KPIs Comparison

4.3 Reinforcement Learning

4.3.1 Batch

4.3.2 LLQP

4.3.3 Others

4.4 Discussion

4.5 Research Contribution

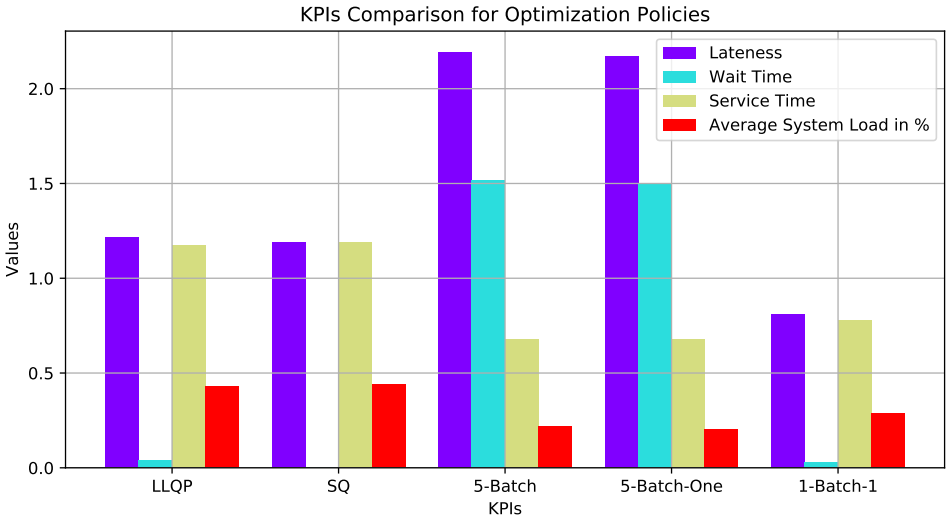

Figure 4.6: KPIs comparison for different optimization policies using the ST solver

Chapter 5

Conclusion

Appendix A

Optimization Results

A.1 K-Batch

A.1.1 KPIs

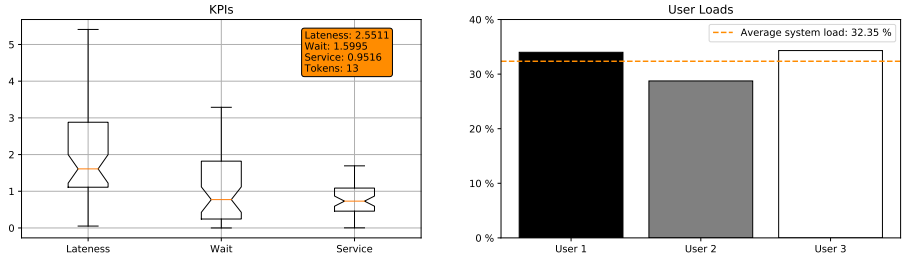

Figure A.1: K-Batch with MSA KPIs

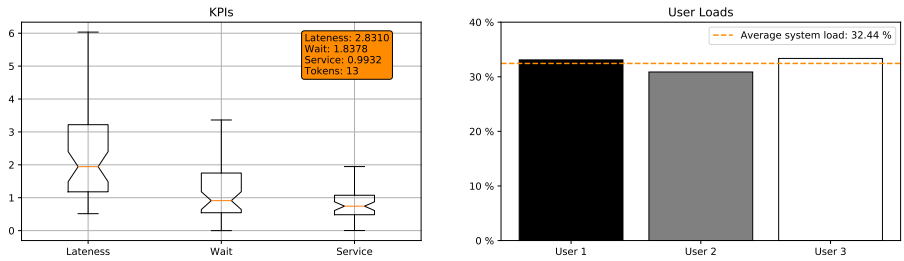

Figure A.2: K-Batch with DMF KPIs

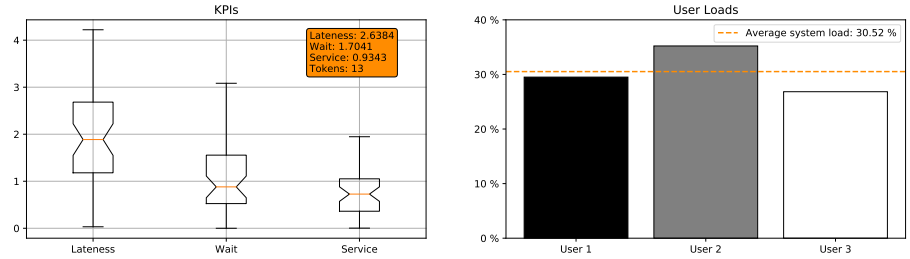

Figure A.3: K-Batch with SDMF KPIs

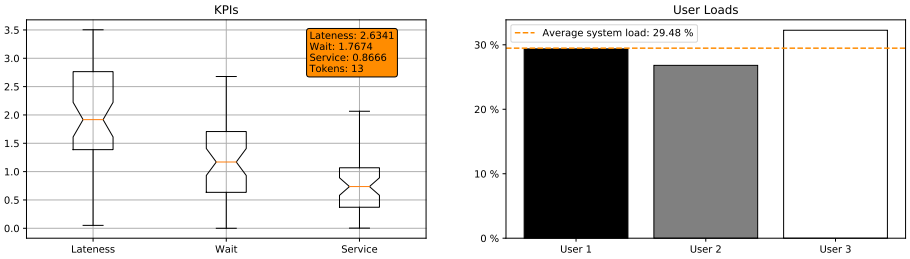

Figure A.4: K-Batch with ESDMF KPIs

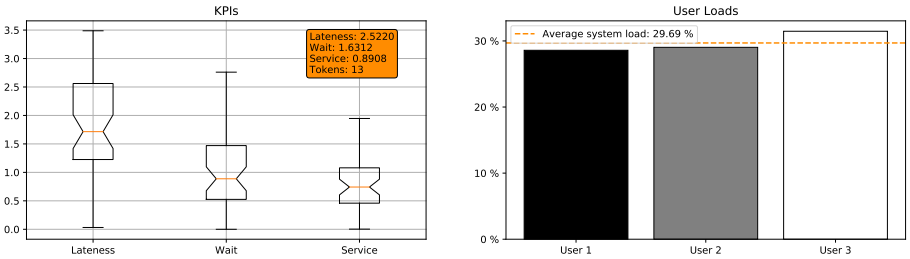

Figure A.5: K-Batch with ST KPIs

A.1.2 Evolution

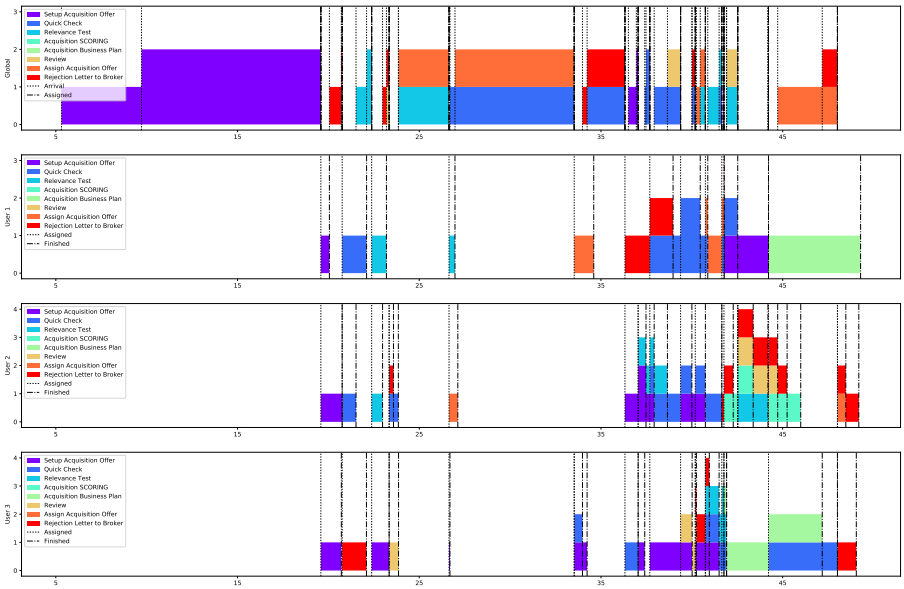

Figure A.6: K-Batch with MSA evolution

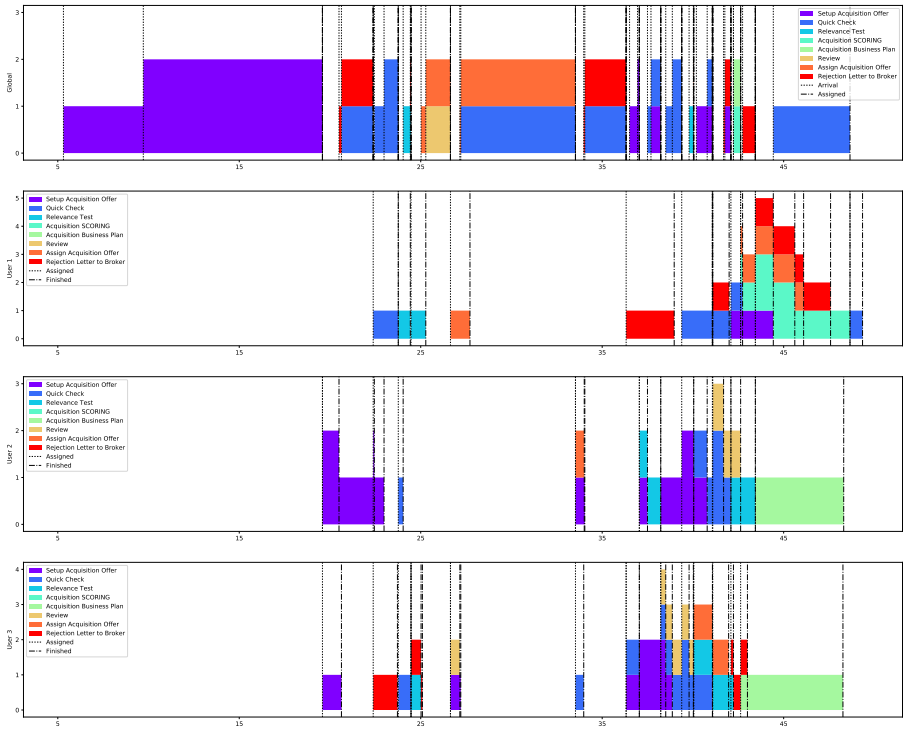

Figure A.7: K-Batch with DMF evolution

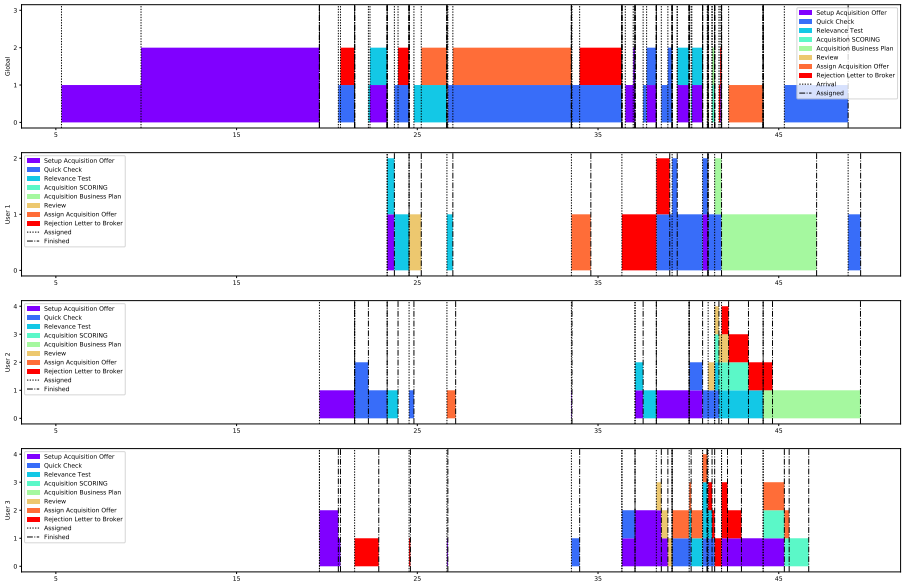

Figure A.8: K-Batch with SDMF evolution

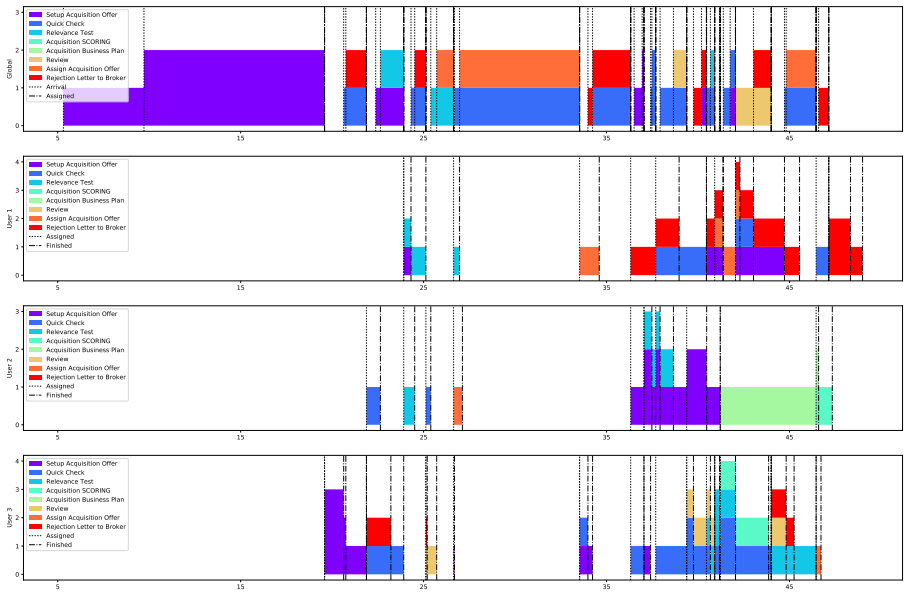

Figure A.9: K-Batch with ESDMF evolution

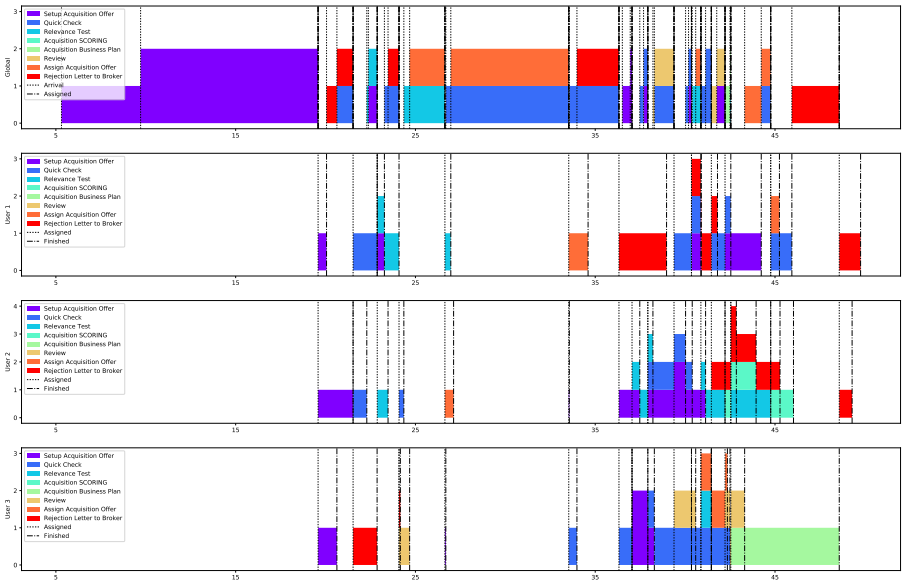

Figure A.10: K-Batch with ST evolution

A.1.3 Batch Sizes Comparison

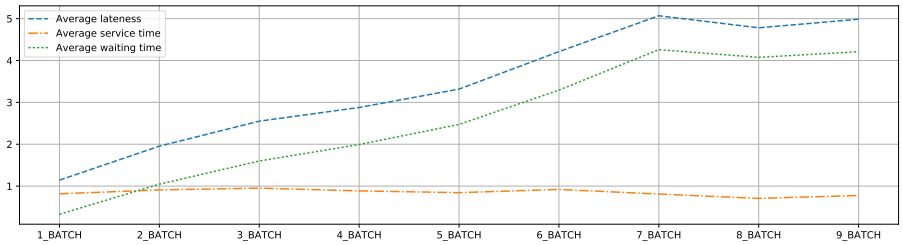

Figure A.11: K-Batch with MSA batch size comparison

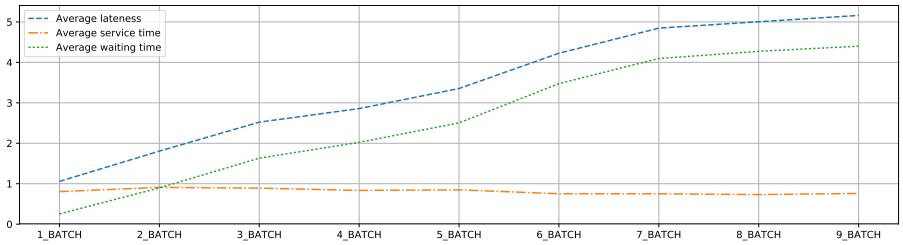

Figure A.12: K-Batch with ST batch size comparison

A.2 K-Batch-One

A.2.1 KPIs

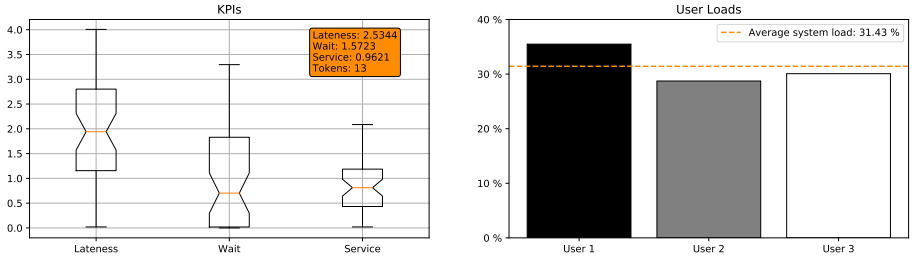

Figure A.13: K-Batch-One with MSA KPIs

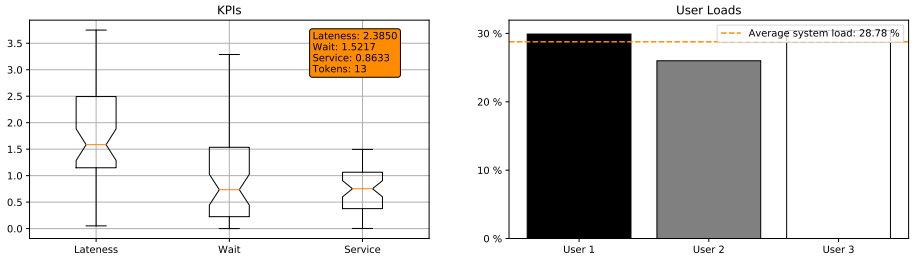

Figure A.14: K-Batch-One with DMF KPIs

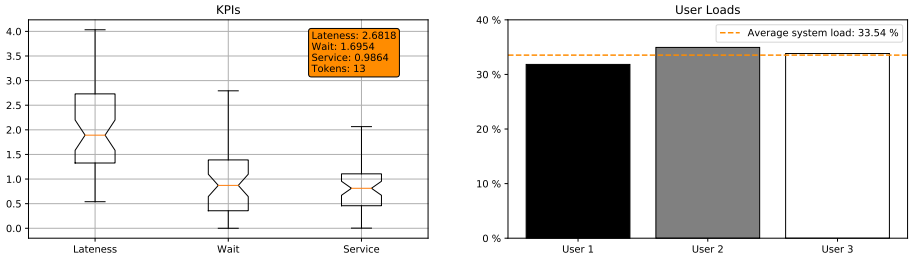

Figure A.15: K-Batch-One with SDMF KPIs

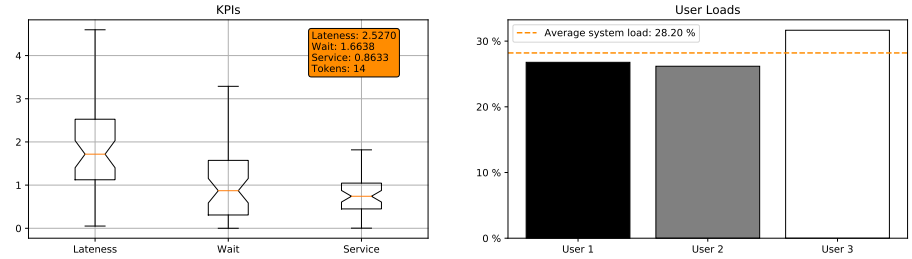

Figure A.16: K-Batch-One with ESDMF KPIs

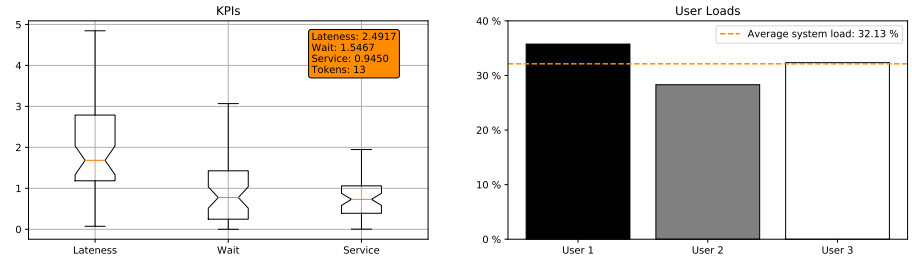

Figure A.17: K-Batch-One with ST KPIs

A.2.2 Evolution

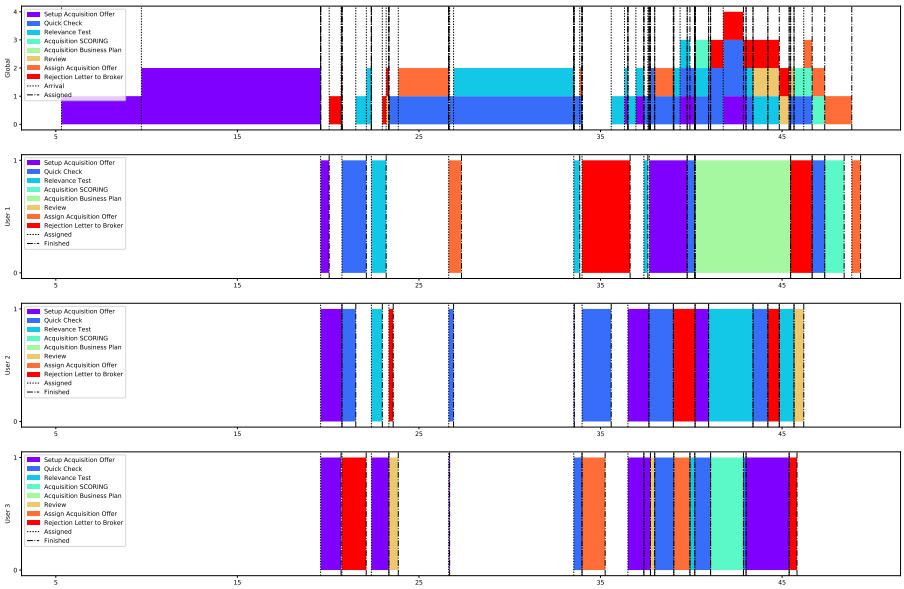

Figure A.18: K-Batch-One with MSA evolution

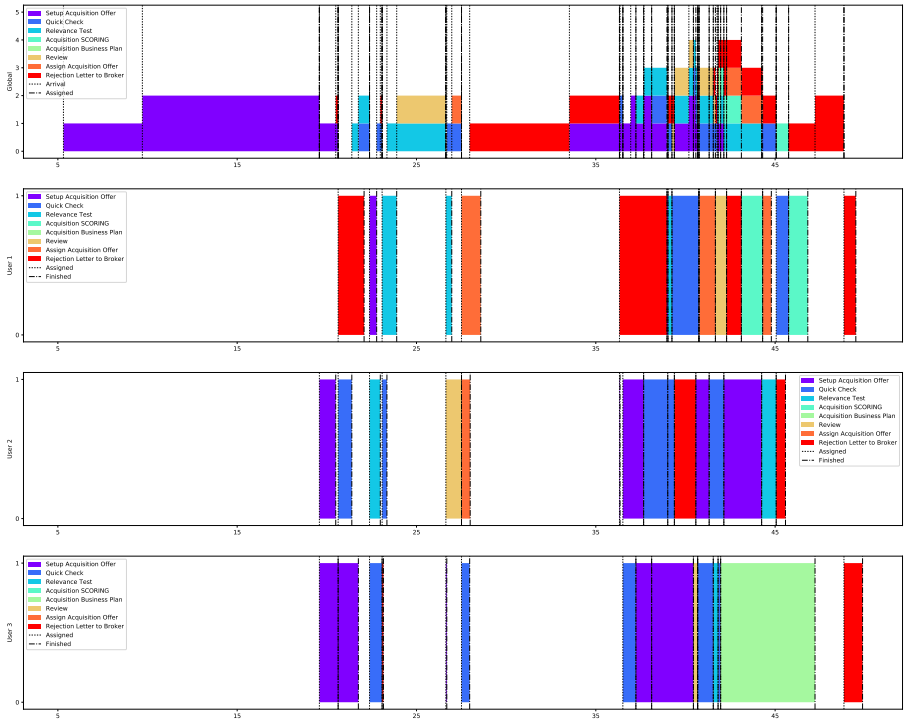

Figure A.19: K-Batch-One with DMF evolution

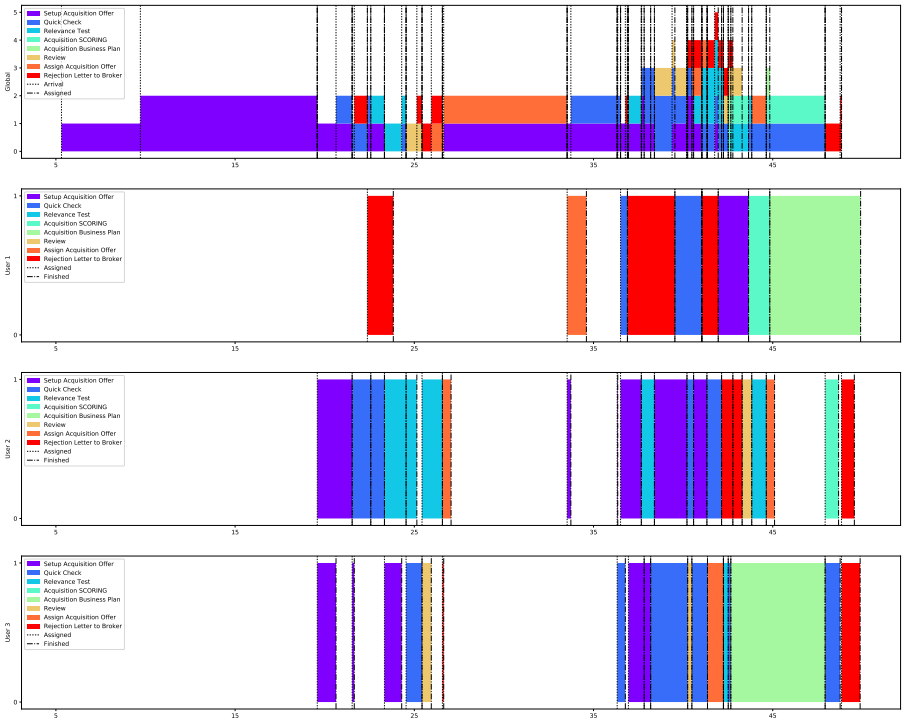

Figure A.20: K-Batch-One with SDMF evolution

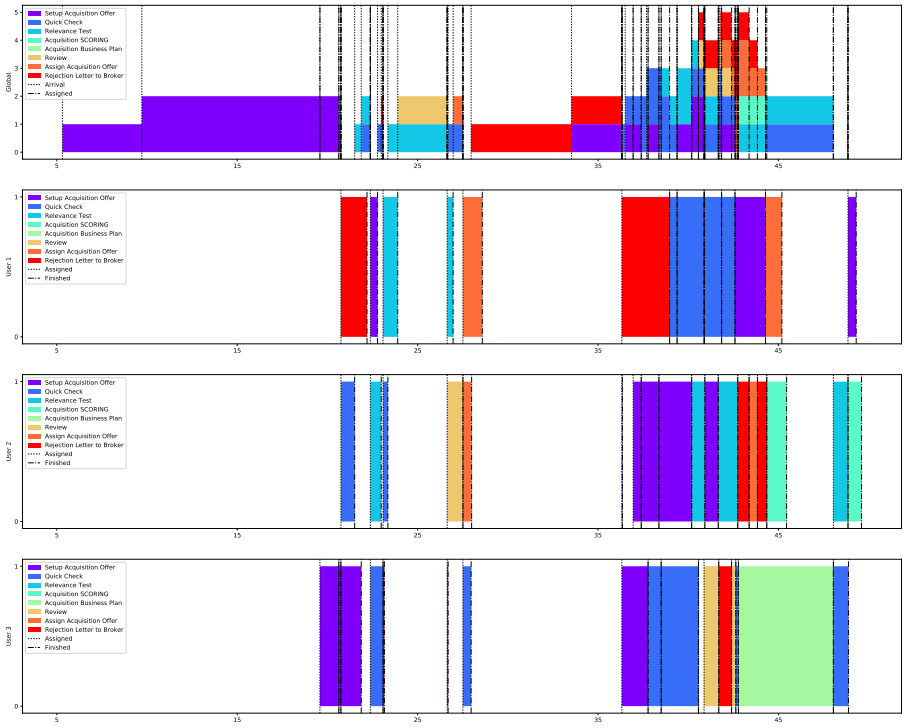

Figure A.21: K-Batch-One with ESDMF evolution

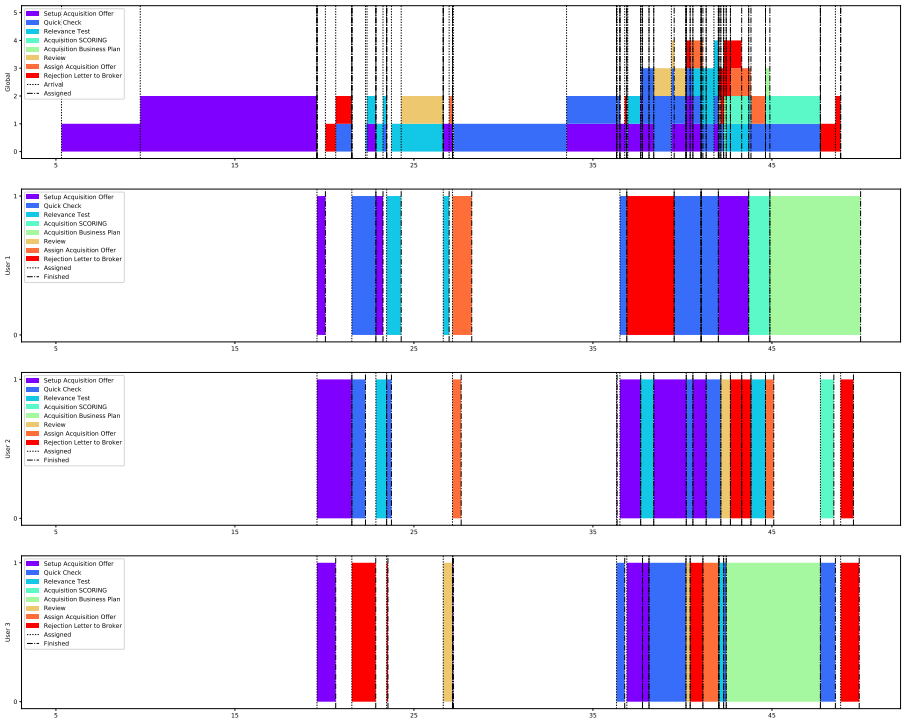

Figure A.22: K-Batch-One with ST evolution

A.2.3 Batch Sizes Comparison

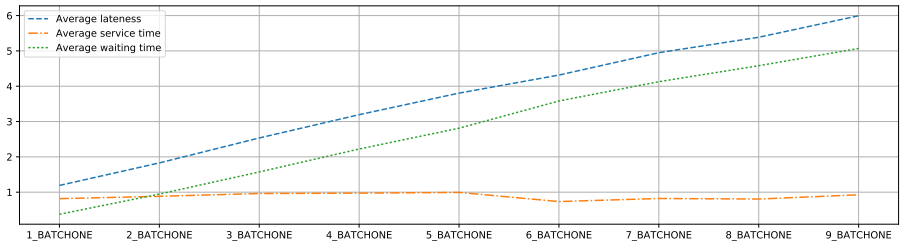

Figure A.23: K-Batch-One with MSA batch size comparison

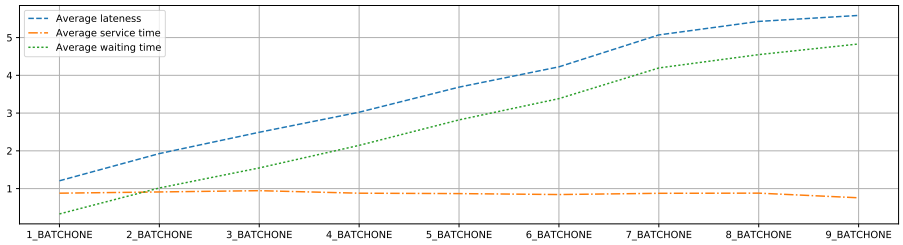

Figure A.24: K-Batch-One with ST batch size comparison

A.3 LLQP

A.3.1 KPIs

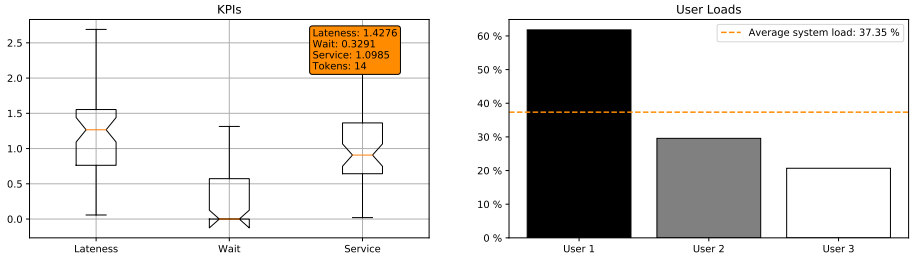

Figure A.25: LLQP KPIs

A.3.2 Evolution

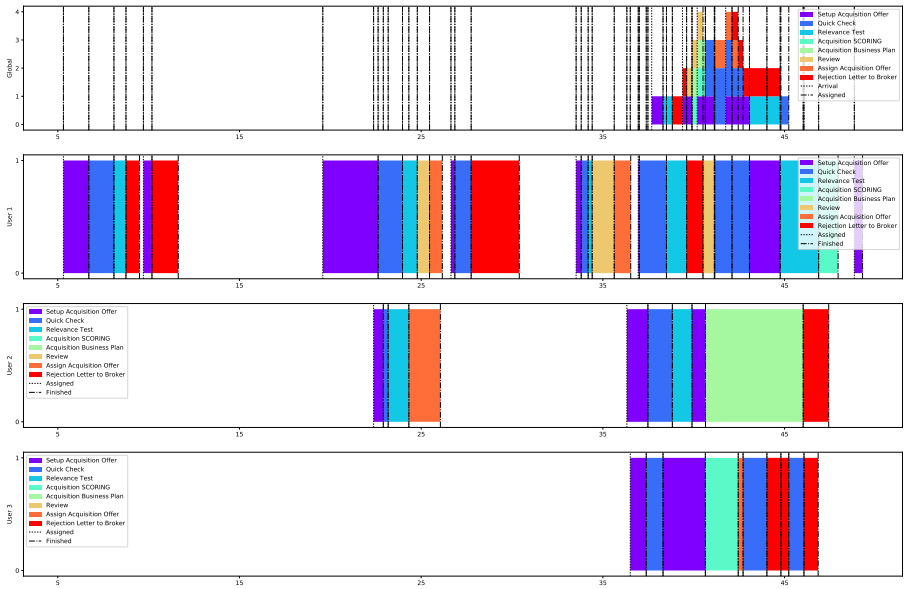

Figure A.26: LLQP evolution

A.4 SQ

A.4.1 KPIs

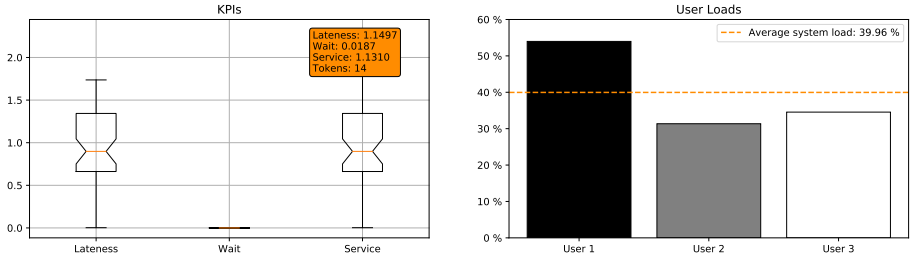

Figure A.27: SQ KPIs

A.4.2 Evolution

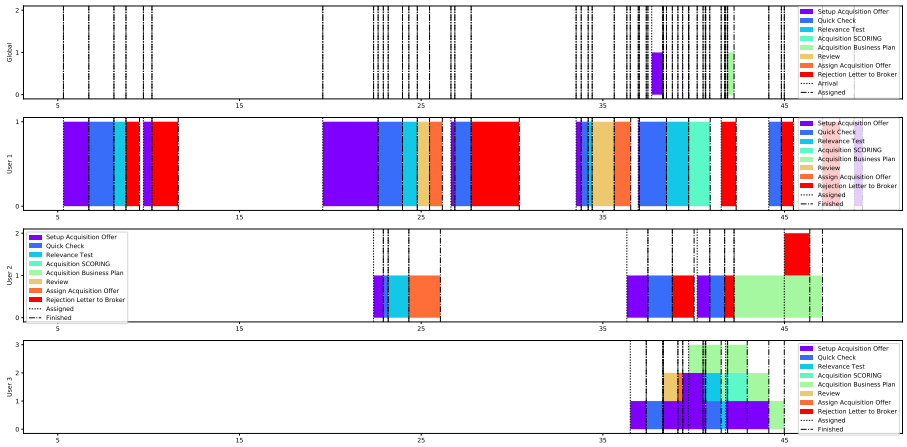

Figure A.28: SQ evolution

Bibliography

[1] Ivo Adan and Jacques Resing. Queueing Systems, March 2016.

[2] Ruth Sara Aguilar-Savén. Business process modelling: Review and framework. *International Journal of Production Economics*, 90(2):129 – 149, 2004. Production Planning and Control.

[3] A. Bahouth, S. Crites, N. Matloff, and T. Williamson. Revisiting the issue of performance enhancement of discrete event simulation software. In *Simulation Symposium, 2007. ANSS '07. 40th Annual*, pages 114–122, March 2007.

[4] K. R. Baker. *Introduction to Sequencing and Scheduling*. John Wiley & Sons, 1974.

[5] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009.

[6] Johannes Bisschop. AIMMS Optimization Modeling, December 2016.

[7] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[8] Dirk G. Cattrysse and Luk N. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, 60(3):260 – 272, 1992.

[9] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *ArXiv e-prints*, November 2015.

[10] Evolved Technologist. BPM Technology Taxonomy: A Guided Tour to the Application of BPM, March 2009.

[11] Shaokun Fan, J. Leon Zhao, Wanchun Dou, and Manlu Liu. A framework for transformation from conceptual to logical workflow models. *Decision Support Systems*, 54(1):781 – 794, 2012.

[12] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[13] Alborz Geramifard, Thomas J. Walsh, Stefanie Tellex, Girish Chowdhary, Nicholas Roy, and Jonathan P. How. A tutorial on linear function approximators for dynamic programming and reinforcement learning. *Found. Trends Mach. Learn.*, 6(4):375–451, December 2013.

[14] Samuel J. Gershman. Empirical priors for reinforcement learning models. *Journal of Mathematical Psychology*, 71:1 – 6, 2016.

[15] George M. Giaglis. A taxonomy of business process modeling and information systems modeling techniques. *International Journal of Flexible Manufacturing Systems*, 13(2):209–228, 2001.

[16] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[17] David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953.

[18] N. Korda and L. A. Prashanth. On TD(0) with function approximation: Concentration bounds and a centered variant with exponential convergence. *ArXiv e-prints*, November 2014.

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[20] A. L. Macintosh. The need for enriched knowledge representation for enterprise modelling. In *IEE Colloquium on AI (Artificial Intelligence) in Enterprise Modelling*, pages 3/1–3/3, Apr 1993.

[21] Norm Matloff. Introduction to discrete-event simulation and the simpy language, February 2008.

[22] Gregory Mentzas, Christos Halaris, and Stylianos Kavadias. Modelling business processes with workflow systems: an evaluation of alternative approaches. *International Journal of Information Management*, 21(2):123 – 135, 2001.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. Letter.

[24] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[25] Michael Racer and Mohammad M. Amini. A robust heuristic for the generalized assignment problem. *Annals of Operations Research*, 50(1):487–503, 1994.

[26] Hajo A. Reijers and Wil M.P. van der Aalst. The effectiveness of workflow management systems: Predictions and lessons learned. *International Journal of Information Management*, 25(5):458 – 472, 2005.

[27] Bruce Silver. *BPMN Method and Style: With BPMN Implementer's Guide*. Cody-Cassidy Press, 2011.

[28] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. Article.

[29] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. *Journal of Machine Learning Research*, 2014.

[30] Andrew James Smith. Applications of the self-organising map to reinforcement learning. *Neural Networks*, 15(8–9):1107 – 1124, 2002.

[31] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[32] Sherry X. Sun and J. Leon Zhao. Formal workflow design analytics using data flow modeling. *Decision Support Systems*, 55(1):270 – 283, 2013.

[33] Sherry X. Sun, J. Leon Zhao, Jay F. Nunamaker, and Olivia R. Liu Sheng. Formulating the data-flow perspective for business process management. *Info. Sys. Research*, 17(4):374–391, December 2006.

[34] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2017.

[35] Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation, 1999.

[36] Peter Trkman. The critical success factors of business process management. *International Journal of Information Management*, 30(2):125 – 134, 2010.

[37] Harry Jiannan Wang and J. Leon Zhao. Constraint-centric workflow change analytics. *Decision Support Systems*, 51(3):562 – 575, 2011.

[38] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.

[39] Daniel D. Zeng and J. Leon Zhao. Effective role resolution in workflow management. 17(3):374–387, 2005.

[40] Kun Zhang and Aapo Hyvärinen. A general linear non-gaussian state-space model: Identifiability, identification, and applications. In *Asian Conference on Machine Learning*, volume 20, pages 113–128. JMLR: Workshop and Conference Proceedings, 2011.