Computational Models for Scheduling in Online Advertising

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Dmitri I. Arkhipov

Dissertation Committee:
Professor Amelia C. Regan, Professor Michael B. Dillencourt, Chairs
Assistant Professor John G. Turner

2016
DEDICATION

To Lydia Arkhipova, Valentina Golovina

and to Ivan Arkhipov, Amelia Regan, Michael Dillencourt, John Turner,
and Claiborne Pell, for making it possible.
TABLE OF CONTENTS

LIST OF FIGURES vi
LIST OF TABLES vii
LIST OF ALGORITHMS viii
ACKNOWLEDGMENTS ix
CURRICULUM VITAE x
ABSTRACT OF THE DISSERTATION xii

1 Online Advertising and Delay 1
   1.1 Introduction .................................................. 1
   1.1.1 One by One Solicitation (OBOS) .......................... 5
   1.1.2 Ad Space Valuation ......................................... 7
   1.1.3 Ad Space Sale Uncertainty and Volume .................. 8
   1.1.4 Traffic ..................................................... 9
   1.1.5 Summary .................................................... 10
   1.1.6 Model Simplifications .................................... 10
   1.2 Review of Related Literature ............................... 11

2 Static Problem Formulation 20
   2.1 Static Problem Description ................................... 20
   2.2 Problem Formulation .......................................... 23
   2.3 Static Optimization Formulation ............................ 25
   2.4 Expectation Relaxation for Stochastic Variables .......... 28
   2.5 A Second Relaxation ......................................... 28

3 Heuristic Solution to the Static Problem 30
   3.1 Solving the Static Problem Heuristically ................. 30

4 Dynamic Problem Formulation 34
   4.1 Markov Decision Process and Dynamic Program Formulation 34

5 Constructing a Policy and Online DP 40
A.1 Proof of Theorem 2.1 .................................................. 101

B Proof of Theorem 2.2 .............................................. 103
  B.1 Proof of Theorem 2.2 ......................................... 103
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Advertiser Filtering</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Algorithmic Advertising Ecosystem</td>
<td>18</td>
</tr>
<tr>
<td>1.3</td>
<td>Ad Polling</td>
<td>19</td>
</tr>
<tr>
<td>1.4</td>
<td>OBOS Execution</td>
<td>19</td>
</tr>
<tr>
<td>2.1</td>
<td>$t^{\text{stop}}$ illustrated</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>State Transitions</td>
<td>39</td>
</tr>
<tr>
<td>9.1</td>
<td>Binary Time Tree</td>
<td>69</td>
</tr>
<tr>
<td>12.1</td>
<td>Advertisers sets of size 10</td>
<td>85</td>
</tr>
<tr>
<td>12.2</td>
<td>Advertisers sets of sizes 11, 12, 13, 14, 15</td>
<td>86</td>
</tr>
<tr>
<td>12.3</td>
<td>XRTDP</td>
<td>87</td>
</tr>
<tr>
<td>12.4</td>
<td>XRTDP-107</td>
<td>87</td>
</tr>
<tr>
<td>12.5</td>
<td>XRTDP-150</td>
<td>88</td>
</tr>
<tr>
<td>12.6</td>
<td>XRTDP-152</td>
<td>88</td>
</tr>
<tr>
<td>12.7</td>
<td>XRTDP-158</td>
<td>89</td>
</tr>
<tr>
<td>12.8</td>
<td>XRTDP-20</td>
<td>89</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Dynamic Program Formalism</td>
<td>38</td>
</tr>
<tr>
<td>11.1</td>
<td>Heuristic Example Results</td>
<td>78</td>
</tr>
<tr>
<td>12.1</td>
<td>Average Percent Deviation Over all Realized Paths, Over all Instances</td>
<td>84</td>
</tr>
<tr>
<td>12.2</td>
<td>Average Net Revenue Over all Realized Paths, Over all Instances</td>
<td>84</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Page</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Grasp_{obos}</td>
</tr>
<tr>
<td>42</td>
<td>Value_{Iteration}</td>
</tr>
<tr>
<td>43</td>
<td>ExecutingTheOptimalOrder</td>
</tr>
<tr>
<td>46</td>
<td>RTDP</td>
</tr>
<tr>
<td>47</td>
<td>RTDP (continued)</td>
</tr>
<tr>
<td>48</td>
<td>ExecutingTheApproximateOrder</td>
</tr>
<tr>
<td>61</td>
<td>UpperBound</td>
</tr>
<tr>
<td>62</td>
<td>Trace</td>
</tr>
<tr>
<td>66</td>
<td>PickPivot</td>
</tr>
<tr>
<td>67</td>
<td>BuildTreeActions</td>
</tr>
<tr>
<td>68</td>
<td>BuildActionTreeIndex</td>
</tr>
<tr>
<td>70</td>
<td>IntervalReduce</td>
</tr>
<tr>
<td>71</td>
<td>BuildTimeTree</td>
</tr>
<tr>
<td>71</td>
<td>BuildTimeTreeIndex</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to acknowledge UCCONNECT, and the Computer Science Chair’s Fellowship for their support. I would like to thank Lars Otten for kindly providing the latex template for this document [32]. I would like to thank the IEEE for permitting work in publication, and accepted for publication to be incorporated into this document. Finally, I would like to thank my committee for their seemingly inexhaustible patience and grace.
CURRICULUM VITAE

Dmitri I. Arkhipov

EDUCATION

Doctor of Philosophy in Computer Science 2016
University of California Irvine

Master of Science Computer Science 2011
University of California Irvine

Bachelor of Science in Information and Computer Science 2009
University of California Irvine

RESEARCH EXPERIENCE

Graduate Student Researcher 2010–2016
University of California, Irvine

Undergraduate Research Assistant 2009
University of California, Irvine

TEACHING EXPERIENCE

Teaching Assistant 2010–2015
University of California Irvine
REFEREEED JOURNAL PUBLICATIONS

IEEE Transactions on Computers

Online Wardriving by Compressive Sensing. 2015
IEEE Transactions on Computers

A Network Option Portfolio Management Framework for Adaptive Transportation Planning. 2011
Transportation Research, Part A: Policy and Practice

Faster converging global heuristic for continuous network design problem using radial basis functions. 2010
Transportation Research Record

REFEREEED CONFERENCE PUBLICATIONS

Yield Optimization with Binding Latency Constraints 2016
International Conference on Soft Computing and Machine Intelligence

UbiFlow: Mobility Management in Urban-scale Software Defined IoT 2015
IEEE Conference on Computer Communications

A Simple Genetic Algorithm Parallelization Toolkit (SGAPTk) for Transportation Planners and Logistics Managers 2015
Meeting of the Transportation Research Board

An on-line data repository for statewide freight planning and analysis 2011
Meeting of the Transportation Research Board
Programmatic advertising is an actively developing industry and research area. Some of the research in this area concerns the development of optimal or approximately optimal contracts and policies between publishers, advertisers and intermediaries such as ad networks and ad exchanges. Both the development of contracts and the construction of policies governing their implementation are difficult challenges, and different models take different features of the problem into account. In programmatic advertising decisions are made in real time, and time is a scarce resource particularly for publishers who are concerned with content load times. Policies for advertisement placement must execute very quickly once content is requested; this requires policies to either be pre-computed and accessed as needed, or for the policy execution to be very efficient. We formulate a stochastic optimization problem for per publisher ad sequencing with binding latency constraints. Within our context an ad request lifecycle is modeled as a sequence of one by one solicitations (OBOS) subprocesses/lifecycle stages. From the viewpoint of a supply side platform (SSP) (an entity acting in proxy for a collection of publishers), the duration/span of a given lifecycle stage/subprocess is a stochastic variable. This stochasticity is due both to the stochasticity inherent in Internet delay times, and the lack of information regarding the decision processes of independent entities.
In our work we model the problem facing the SSP, namely the problem of optimally or near-optimally choosing the next lifecycle stage of a given ad request lifecycle at any given time. We solve this problem to optimality (subject to the granularity of time) using a classic application of Richard Bellman’s dynamic programming approach to the 0/1 Knapsack Problem. The DP approach does not scale to a large number of lifecycle stages/subprocesses so a sub-optimal approach is needed. We use our DP formulation to derive a focused real time dynamic programming (FRTDP) implementation, a heuristic method with optimality guarantees for solving our problem. We empirically evaluate (through simulation) the performance of our FRTDP implementation relative to both the DP implementation (for tractable instances) and to several alternative heuristics for intractable instances. Finally, we make the case that our work is usefully applicable to problems outside the domain of online advertising.
Chapter 1

Online Advertising and Delay

1.1 Introduction

In online advertising when a web user’s browser requests an advertisement from a publisher’s content server that server then sends an ad request to the publisher’s ad server. The publisher’s ad server may then use an ad network as a broker to sell the available ad space to advertisers. The ecosystem is fairly complex and involves multiple interactions with varied agents, often with competing interests. Figure 1.2 provides a schematic representation of some of the complexities of the environment and some transactions found therein. We have marked in bold the path we are interested in exploring in this paper, in particular the cycle defined by edges 9-12.

The added value that the ad network provides is access to its in-network advertisers. In-network here means that the ad network has an established programmatic communication channel with this set of advertisers. Those web content servers that use an ad network are referred to as in-network publishers. Thus, an ad network brokers ad sales between in-network publishers and in-network advertisers in real-time. The contract between the ad
network and each in-network advertiser specifies the price at which the in-network advertiser will buy ad space but does not require an advertiser to buy each ad space offered to it by the ad network.

A notable and well studied aspect of the online advertising are filtering requirements between ad spaces and the ads that may fill them. Such requirements are generally dependant on the attributes of the consumers viewing the published content e.g. demographic information such as age and gender, on the attributes of the publisher who’s content is being consumed, and on the attributes of a given advertiser. Filtering checks may be done at the time the user begins to load publisher content, or may be approximated based on the historic user access statistics available for a publisher. In figure 1.1 we give an illustration of how such filtering may progress looking at the dimensions of age and gender. Our illustration hints at the exponential number of the feature sets which complicates filtering on large numbers of attributes. In our work we do not deal with the filtering problem, rather our work takes place after appropriate filtering has concluded, and we have a set of acceptable candidate advertisers.

![Figure 1.1: Advertiser Filtering](image-url)
In this paper we use the term “advertiser” loosely, in our use it references an entity which is interested in purchasing ad space at a deterministic price. In industry terms this is most directly applicable to “preferred contracts”, but the term can be made to apply also to ad space sales on a real time bidding (RTB) platform, or to “direct sales”. Indeed Balseiro et al [6] give several templates for the reduction in a related problem. The reductions described by Balsero sacrifice optimality and yield by a discrete factor, however as in our work we already make the analogous sacrifice with respect to time discretization the reductions are directly applicable here. The reader of this work is perhaps best served by thinking of an “advertiser” simply as a server which decides to purchase or not purchase ad space on solicitation with the knowledge that the situation generalizes usefully to more complex arrangements that exist in the industry.

The contract between the ad network and each in-network publisher specifies the proportion of the revenue (usually specified in cost per mille (CPM)) earned by the ad network in selling the ad space to be filled by content generated and served by the in-network publisher to any in-network advertiser.

The contract between the ad network and each in-network publisher further specifies the maximal allowable servicing-time for each ad request. Here servicing-time refers to the time spent in finding an advertiser willing to buy the ad space indicated by the ad request. This constraint ensures a guaranteed content load time. The ad network imposes a formula for calculating penalties for service-times exceeding the deadlines; the size of the penalty is positively related to the lateness of the service time of the ad request.

When a sufficiently long delay is detected, the ad network will return a “house-ad” thus terminating the ad request. A house-ad is an advertisement for which the ad network is not paid; it is an ad that is generated as a fall back option. Generally, a house-ad advertises the ad network itself (for example “we can sell your ad here”, or a service owned by the same parent company) and the value of deploying a house-ad is considered to be negligible.
Without loss of generality we assume that deploying house-ads will generate no profit for the ad network.

Displaying a house-ad indicates the failure to find a willing in-network advertiser to purchase a given ad request in the allotted time. This means in effect, that the ad network purchased the ad space. However, we assume that any loss associated with this transaction is negligible and assign it a value of 0 to simply indicate the lack of revenue.

The penalty for exceeding the maximum allowable servicing-time is large enough to ensure that extending service time of the ad request beyond this point will reduce the revenue earned in the event of success. Two classes of applicable penalty functions are:

1. A constant penalty for exceeding the deadline: The penalty is paid regardless of whether the last solicited advertiser has responded.

2. A non-constant penalty for exceeding the deadline: Paid immediately after the last solicited advertiser responds.

A constant penalty is consistent with the policy of showing a house ad immediately as the deadline is exceeded. A non-constant penalty is consistent with always waiting for the last solicited advertiser to respond. A non-constant penalty can be charged by the publisher (explicitly or implicitly) for introducing delays and lowering the web user’s experience. A non-constant penalty function can thus ensure that a small delay shouldn’t be penalized excessively, but a large delay can be prohibitive, so the ad network will reject solicitations that are likely to incur such a prohibitive penalty. In this work we develop an algorithm with bounded sub-optimality based on the case of a non-constant penalty function, under assumptions common in the revenue management literature.
1.1.1 One by One Solicitation (OBOS)

The protocol between the ad network and the in-network advertisers is very simple. When a web-user’s browser requests data from an in-network publisher, this publisher, in turn requests an ad from the ad network. The ad network receives the request and filters out all in-network advertisers from whom it does not expect to receive a reply quickly enough to honor the service time contract between the ad network and the publisher. The remaining set of advertisers are known as the set of feasible in-network advertisers. The ad network will then sequentially poll its in-network advertisers in some order with all identifying information regarding the ad request, the associated ad publisher, ad space and web-user. We use the terms “sequential polling”, “one by one polling”, and “one by one solicitation” or (OBOS) interchangeably. Each in-network advertiser responds with a “yes or no” message to indicate purchase or non-purchase of the ad request (if purchase is indicated the funds are transferred immediately). After each unsuccessful poll the polled advertiser is removed from the set of feasible advertisers. Some feasible advertisers may become “certainly unprofitable” after an attempt. Because after each attempt the amount of time before the contract deadline expires diminishes, some tasks while still feasible (because they have not yet been attempted) will be “certainly unprofitable” as even were the task to succeed the net revenue derived from the task would be non-positive (at some time the costs of attempting a given advertiser-attempt will out-weigh the benefits of its success even if the successful task execution experiences no delay).

The protocol terminates when either one of the polled advertisers agrees to buy the ad space or the ad network finds that none of the remaining unpollled advertisers are feasible and “potentially profitable”. In the case where the ad space was bought, the space is filled with the purchasing advertiser’s ad, and the published content is transmitted to the consumer. In the case where no feasible “potentially profitable” advertisers remain the ad network fills the ad space with a house ad.
Figure 1.3 illustrates these interactions.

An in an instance of OBOS execution 0 or more failed solicitations occur, followed by a successful solicitation (or a house-ad). We illustrate such an execution in figure 1.4. In our illustration 4 unsuccessful solicitations are made, terminated by either a successful solicitation, or a house-ad.

The situation in which two advertisers are assigned to the same ad request is forbidden. This constraint indicates that we may not solicit an ad request to multiple in-network advertisers simultaneously. In fact we must solicit in-network advertisers sequentially, advancing to the next advertiser in the sequence only when the previous advertiser’s rejection is received. Of course once an advertiser purchases the ad request we no longer solicit other advertisers to purchase the same ad request. The greater part of the processing time for an ad request solicitation is spent at the advertiser as it processes the request, decides on a response, encodes and transmits the response. Thus modifying the order in which the solicitations occur can greatly alter the full delay experienced by the user in loading the full publisher page.

Within the context of serving programmatic advertising there are two predominant programming models, real time bidding “RTB” platforms or “waterfall” platforms, executing the protocol we have termed “OBOS”. For a given ad request an ad network will use execute either “RTB” or “OBOS” in order to sell the ad space represented by the ad request.

In industry, the sequence in which advertisers are solicited for a particular ad space is called a “waterfall” or “daisy chain”, and the process of creating the waterfall is known as “yield optimization”. The process of executing OBOS as we have defined it is known within the industry as “executing synchronous ad calls”, “daisy chaining ad tags”, or “falling through a waterfall”.

6
The problem of finding the optimal (with respect to expected value) sequence to solicit the advertisers in is a difficult challenge. Examining each possible sequence would require $N!$ operations (assuming that $N$ is the number of in-network advertisers). Clearly this is too computationally expensive to be considered except for very small values of $N$. In this work we explore less computationally expensive methods for determining this optimal sequence, and a heuristic method that performs very close to optimal (in our tests it was two percent worse, on average, than the optimal policy).

In real time bidding an RTB platform solicits multiple advertisers simultaneously and chooses the highest bidder who’s bid arrived within the time limit; the winner will then pay the price bid by the second-highest bidder (a sealed-bid second-price auction).

In an RTB system:

- Advertisers must be able to assign a bid value for each ad space.
- Advertisers will not know at the time they bid whether their bid was successful or not.

1.1.2 Ad Space Valuation

Dynamically assigning a bid value for a given ad space is not an easy problem. If a bidder is bidding their true valuation as is assumed in second price auctions then the seller has the opportunity to incrementally learn the buyer’s valuation behavior when repeated auctions for similar items occur, and the same sets of buyers and sellers take place.

When auctions are repeated for non-unique goods there is a high probability that similar interactions will occur. In cases where repeated auctions for similar goods are occurring it is understood that second price auctions are not incentive compatible, and sub-optimal. In this situation buyers bidding their true valuations is not an equilibrium. Because truth telling
is not an equilibrium, advertisers find it difficult to determine what price to bid on a given impression.

Consequently, many advertisers choose to fix a price per fiscal quarter with the ad network and pay this price each time an acceptable ad impression comes in. The advertiser only needs to accept valuations where the estimated value of the request is at least as high as the contract price.

From a system wide perspective when prices are agreed upon in advance it is easier for the ad network to prioritize who to send requests to and achieve a near optimal match between publishers and advertisers as compared to sending requests to all valid advertisers.

1.1.3 Ad Space Sale Uncertainty and Volume

Advertisers face pacing constraints for their ad campaigns. The pacing of an advertising campaign specifies the rate at which funds dedicated towards the campaign are to be spent. How closely the true amount spent matches to the intended amount as specified in the pacing for a campaign is an important measure of the quality of the ad network facilitating ad space sales. In industry terms the ad network is acting as a “Demand Side Platform” (DSP) with respect to advertisers or “creatives”, and as a Supply Side Platform (SSP) with respect to publishers. Another important measure is the total number of ad spaces filled for a given campaign. For any given campaign, an advertiser will prefer an ad network which can maximize the number of appropriate ad spaces filled, while keeping close to the pacing of the campaign.

Even if a bid price for a particular ad space were easy to determine, an advertiser would be uncertain of the success of their bid, and thus would be unable to adjust future bid pacing quickly. As campaigns deviate from the required pace the bid amounts can increase (if the
amount spent is too low) such increases can result in bidding higher than would be necessary to acquire ad space and thus decrease the total amount of ad spaces filled by the campaign. The difficulty of properly determining a bid price for an ad space increases the uncertainty of whether a particular RTB bid is successful.

In practice ad networks will often give up the potentially higher prices available from an RTB for the certainty and greater volume offered by a fixed price ad space purchaser. Greater predictability removes some of the uncertainty associated with consistently maintaining campaign pacing.

To fully remove the variability an advertiser may negotiate directly with publishers to buy from them guaranteed ad space. In this arrangement a fixed price, a fixed number of ads, and a fixed pace for ad space sales is established between a publisher and an advertiser. In many ways this arrangement would suit both parties, however a given publisher cannot reasonably guarantee either a fixed number of visitors, or the pace of the visitors arrival at the publishers page.

An ad network, which receives ad requests from blocks of publishers is better positioned to guarantee both quantity and pace.

1.1.4 Traffic

Finally, an added benefit of the OBOS approach over RTB is that the number of messages broadcast by the ad network is significantly reduced. Where in RTB a message is sent and received between the ad network and all applicable advertisers, in OBOS these messages are exchanged sequentially until an ad request is filled successfully, so with respect to the number of messages exchanged RTB is the worst case of OBOS. In other words because prices are known in advance, determining a market clearing price is not necessary.
1.1.5 Summary

In summary there are several reasons why one-by-one polling with a fixed price may be preferable to an auction mechanism:

- Bid price selection is handled periodically and so does not need to be determined dynamically.
- There are smaller deviations from pacing goals than with RTB.
- Fewer network messages are generated.

1.1.6 Model Simplifications

The protocol described above diverges slightly from the situation as it arises in industry. In point of fact ad networks might solicit even advertisers they do not expect to respond in time. Ad networks will solicit advertisers until the service time negotiated with their publisher has nearly elapsed and at that time if the ad space has not yet been purchased the ad network will display a house ad. Doing so leads to a troubling situation where an advertiser may be in the process of deciding whether or not to purchase ad space when the ad space expires and is no longer available for purchase. A decision to purchase the ad request at such a moment will result in the inability of the ad network to fulfill the purchase. While this is problematic, it is far less frequent a situation than would result from allowing publishers simultaneous consideration of ad spaces.

In the model that we develop in this paper we do not model this complication directly. Rather, we assume that the ad network must wait for each polled advertiser to respond before either soliciting another advertiser or serving ad content (whether a house ad, or an advertiser ad) to the publisher. This means that our model ad network may violate the
response time constraint by an arbitrary amount of time. We combat this simplification by specifying that after each advertiser responds the ad network will leverage available data to remove those advertisers who would be likely to violate the response time constraint. We also assign a cost function that is monotonically increasing with time and if the deadline is exceeded we deduct this cost from any profit generated by an ad space sale. Our general model of the cost function makes it possible to assign arbitrarily high costs to violations of the response-time constraint.

Our problem formulation is written to reflect the general case of probabilistic mutually exclusive task sequencing, rather than the specific motivating problem. As such we use the terminology of ordering a set of mutually exclusive tasks in a fixed time budget. For ease of explanation, in our problem description and later in our dynamic programming formulation, as closely as possible we use Powell’s modeling notation from chapter 5 of [33].

1.2 Review of Related Literature

Though relatively recent, web advertising is a complex and studied field. In this dissertation we examine one small component of the web advertising ecosystem, and in this section we review the work most related to our own. Feichtinger, Hommes, and Milik give an early survey of modeling work in this space [18]. Readers with a broader focus will find Kurula et al’s more recent survey [25] of direct advertising interesting, Muthurishnan’s excellent characterization of the mechanics of ad exchanges [30] is also an important resource. Wang and Yuan [43] provide an introduction and tutorial to the real time bidding (RTB) model, and Yuan et al [44] provide a detailed analysis of a real RTB system.

A variety of works address different aspects of the online advertising ecosystem. Chickering and Heckerman [14] address the tradeoff between satisfying contracts specifying impression
counts per time period while simultaneously maximizing the expected number of ad clicks. Ad clicks generate more revenue to the publisher but reserve contracts are often specified in impression counts. Ghosh et al [20] consider the problem facing an advertiser bidding agent in meeting pacing goals while minimizing costs and purchasing high quality placements. The researchers approach consists of learning the distribution from which ad impressions are being drawn. Next their method leverages this learned knowledge to bid appropriately to satisfy pace and quality constraints while minimizing net expenditure. Devanur and Hayes [17] consider the problem facing a search engine in auctioning off keyword triggered ad placements while not exceeding advertisers budget constraints. Devanur and Hayes solution proceeds in two phases similarly to [20], though in a different setting. In the first phase the distribution of keyword lookups is learned, next the learned distribution is used to price keyword placements with bounded suboptimality. In [42] Vee et al achieve a near optimal online ad allocation using a two-phase scheme to compute an allocation plan based on sampled data. The allocation plans are then used to assign impressions online to advertisers with compatible targeting rules. Salomatin, Liu, and Yang [36] propose a framework to optimize search engine revenue per keyword query while meeting guaranteed delivery contracts, and having access to ad auctions.

Balseiro, Besbes, and Weintraub [5] establish a mechanism for optimally setting reserve prices on ad spaces in real time, while meeting all guaranteed contracts. Turner [40] analysed the problem of satisfying advertiser reach and frequency constraints under audience restrictions per advertiser. The problem examined in [40] is considers constraints important to advertisers and their goals. Specifically, this work optimizes advertisement allocation as a stochastic optimization problem where ad space availability is a Poisson random variable, and web user classifications determine whether a given ad space may be assigned a particular ad. When analysed from the perspective of an advertiser user quality considerations and validity of a given ad space to a user is a more pressing concern than round trip or load time. In our work we model the decisions made at the ad server in real time as a Bernoulli random variable,
and focus instead on the time taken to make this decision. When the ad allocation problem is analysed considering constraints important to the publisher, or a publisher oriented ad network content load time becomes a more important consideration and the fine details of the decision process at the ad server are not known. Hojjat and Turner’s subsequent work [22] also analyses guaranteed ad allocation considering constraints important to advertisers. In this later work the researchers discovered that column generation is an appropriate technique for quickly producing user-level ad patterns under reach, frequency, spread, pacing, and variety constraints. This important work considers optimal pattern construction for the problem facing publishers satisfying guaranteed contracts under the above mentioned considerations. Our work can be seen as constructing patterns (sequences in our terminology) given constraints important to the publisher (load time constraints). While both [40] and [22] consider satisfying guaranteed delivery contracts in our work we consider non-guaranteed contracts, another important contract class.

Balseiro et al [6] consider the problem of maximizing long-term revenue while satisfying traditional reserved contracts with high quality ad spaces. Balseiro et al are able to achieve an optimal result for any prespecified preference of ad quality allocated to reserve contracts versus maximizing short term revenue. In the joint optimization model developed by the researchers the long-term revenue gained from successfully satisfying long-term reserved contracts with high quality ad placements is balanced with the greater short term revenue available from selling an ad on an ad exchange. Publishers seek to maximizes their long-term profit, which is a function of (1) traffic volume, and (2) the advertising revenue made per arrival. Najafi-Asadolahi and Fridgersdottir [31] have modeled this trade off explicitly for cost-per-click ads, and give a closed form solution for the publisher’s pricing problem. Our work concentrates instead on short term planning with fixed prices, but stochastic delays.

Lang et al [26] give efficient algorithms for selecting from valid demand-supply paths through a network of real-time bidding systems. Lang et al’s work effectively addresses the problem of
finding revenue valid, optimal trades ad space trades under conditions where advertisers are assumed to accept valid ad placements. In this our work we do not make this assumption but rather model the acceptance and rejection of ad space as a choice made by the advertiser’s ad server (these choices can be modeled as random variables). Another aspect where our approach differs from Lang et al’s approach is that they consider a valid path as one in which certain filtering constraints are satisfied whereas we assume that filtering has occurred prior to the optimization. Our assumption is based on the specification and the speed of adoption of the OpenRTB standard [35] which standardizes the determination of advertiser-publisher validity and in many cases removes the need for explicit querying. Our approach places greater importance on timely delivery guarantees than [26], this is again consistent with guarantees made explicit in the tmax field of the OpenRTB standard [35]. Finally, Lang et al [26] treat an RTB auction as the primary sales mechanism employed by ad networks in selling ad space; in fact it is often the case that an ad network will treat its set of in-network advertisers preferentially and will choose when possible to sell ad-space to them directly in the manner discussed in section 1.1. Selling ad space directly to an in-network advertiser can often be done more quickly than participating in an ad auction, and avoids the revenue sharing discussed in [26].

Our approach places greater importance on timely delivery guarantees than [26], this is again consistent with guarantees made explicit in the tmax field of the OpenRTB standard [35]. Finally, Lang et al [26] treat an RTB auction as the primary sales mechanism employed by ad networks in selling ad space; in fact it is often the case that an ad network will treat its set of in-network advertisers preferentially and will choose when possible to sell ad space to them directly in the manner discussed in section 1.1. Selling ad space directly to an in-network advertiser can often be done more quickly than participating in an ad auction, and avoids the revenue sharing discussed in [26].
Chakrabory et al [13] analyzes a related problem from the perspective of an ad exchange; in their work they develop an optimization framework accounting for both allocation (of ads to ad networks) and solicitation (which ad networks to solicit for a bid on ad request). This paper while related to [13] differs from their model in several ways. Chakraborty et al choose to model the bandwidth constraints of an auction (ad exchanges wasting bandwidth on bid solicitations that are unsuccessful are themselves less successful) by these constraints they intend to tighten solicitation selectivity (those ad networks more likely to win an ad request are more likely to be solicited with it. Of course their model has the additional objective of optimizing revenue. Chakraborty et al’s model implicitly (and reasonably) assumes that conservation of bandwidth will translate to a lower latency system (thus ensuring timely delivery). Our model addresses the problem encountered by an ad network rather than an ad exchange. The distinction between the two entities is that an ad network usually attempts to service a set of in network advertisers prior to (or selectively for certain ad requests) using an ad exchange for running an auction. The protocol between the in network advertisers and the ad network (described in section 1.1 ) specifies that solicitations must occur sequentially, rather than in an ad auction format (wherein solicitations are made synchronously). In our work (in contrast to [13]) we address the service time constraints of the problem explicitly while asserting that sequential solicitation will naturally use less bandwidth than synchronous solicitation. We think that modelling service time constraints is a reasonable choice as those constraints are explicitly encoded in the OpenRTB standard [35] and thus their violation could arguably be a breach of the agreement between the ad network and the in network publishers.

Our model addresses the problem encountered by an ad network rather than an ad exchange. The distinction between the two entities is that an ad network usually attempts to service a set of in network advertisers prior to (or selectively for certain ad requests) using an ad exchange for running an auction. The protocol between the in network advertisers and the ad network (described in section 1.1 ) specifies that solicitations must occur sequentially,
rather than in an ad auction format (wherein solicitations are made synchronously). In our work (in contrast to [13]) we address the service time constraints of the problem explicitly while asserting that sequential solicitation will naturally use less bandwidth than synchronous solicitation. We think that modelling service time constraints is a reasonable choice as those constraints are explicitly encoded in the OpenRTB standard [35] and thus their violation could arguably be a breach of the agreement between the ad network and the in-network publishers.

Balseiro et al [6] analyse the problem of satisfying publisher contacts (in their case specifying impression counts per publisher, and impression quality constraints) while also maximizing revenue from ad space sales on an ad-exchange (an RTB platform). The work of these researchers uses similar methods to our own, and analyse a highly related problem. In the case of their research, quality variables as well as satisfying ad space quantities per publisher are a fundamental object of study. We have found that more recently, ad space availability has not been a binding constraint as ad space has become plentiful. Rather we find that timely delivery has become a critical concern for all entities in the programmatic advertising ecosystem. Our work studies the impact of timing on the micro-transactions presumed to occur in [6].

Mirrokni and Nazerzadeh[29] describe and analyse several families of preferred deals in comparison to reservation contracts and the RTB mechanism. Their work develops a method for constructing approximately optimal (in terms of publisher revenue) preferred deals. In this paper we discuss the sub-class of preferred deals characterized as \((\rho, \mu = 0)\) (in their lexicon), that is prices per publisher, advertiser pair are fixed, and the advertiser is under no obligation to purchase any of the offered impressions. They demonstrated that the use of this class of preferred deal results in reduced revenue for the seller when compared to the use of an RTB mechanism with a buyer reserve price. However, this class of contract is not uncommon. Impression buyers can be interested in this class of deal because it pro-
vides a guaranteed impression channel, with a known price, and predictable impression rate; this is valuable when seller valuations fluctuate due to deviations from target pacing. The algorithm proposed for choosing the next buyer to solicit for a general preferred deal is at least linear in the number of advertisers, which is problematic for cases where the number of applicable advertisers is small, and timing constraints stringent. Our work explores this class of contract, with seller imposed round trip time constraints not included in the analysis of Mirrokni and Nazerzadeh.

Amiri et al’s work on developing advertisement schedules through a Lagrangian relaxation of the ad placement problem [2] is perhaps most related to our work in this paper in that their static orderings can be implemented easily either on ad servers or on web user’s browsers. However, their formulation also does not explicitly account for delay constraints or delay stochasticity.

Dean, Goemans, and Vondrak [16] achieve a bounded sub-optimality for both static and adaptive policies on a stochastic knapsack variant related to the problem we address in this paper. In their work the authors present several policies with different sub-optimality for the variant of the stochastic knapsack in which item value is predetermined, but item weight is random and realized only when an item is selected. This variant of the stochastic knapsack maps directly to the problem addressed in our work when the non-convex function we reference is used. The sub-optimality bounds of [16] do not apply to variants of the problem where a convex function is used, and we contribute a solution algorithm with bounded sub-optimality for this case in this work.
Figure 1.2: Algorithmic Advertising Ecosystem
Feasible, “potentially profitable” advertisers are those advertisers who have not been solicited with an ad-request the ad network expects to respond quickly enough that the ad poll is expected to be profitable.

Figure 1.4: OBOS Execution
Chapter 2

Static Problem Formulation

2.1 Static Problem Description

At a time $t = 0$ we are assigned a set of $N$ unique mutually exclusive tasks: $\mathcal{A} = \{1, ..., N\}$. We are given a time budget/deadline time $t^{\text{max}}$. We are constrained to successfully complete no more than 1 of these tasks. We may attempt tasks sequentially but not in parallel. We define a convex piece-wise linear function $c^1(t^{\text{max}}, t)$ that gives the cost associated with finishing an attempt at any task after $t$ time units have elapsed (since the tasks were assigned). This cost $c^1(t^{\text{max}}, t)$ will only grow after $t > t^{\text{max}}$ (prior to which there is no penalty) and it will grow as a convex function of $t^{\text{max}} - t$.

$$
c^1(t^{\text{max}}, t) = \begin{cases} 
0 & : t \leq t^{\text{max}} \\
c^2(t^{\text{max}} - t) & : t > t^{\text{max}} 
\end{cases} \tag{2.1}
$$
The cost function $c^1$ is defined in terms of another function $c^2 \nu(t^{\text{max}} - t)$, which represents the penalty associated with exceeding the deadline by $t - t^{\text{max}}$ time units (assuming $t > t^{\text{max}}$). The policy may be problem dependent but for the purposes of our current work we use the policy:

$$c^2(t^{\text{max}} - t) = \nu(t^{\text{max}} - t)^2 \quad (2.2)$$

Each task attempted will either succeed or fail. Task $a$, if attempted will succeed with some known probability $p_a$. Task $a$ will fail with probability $1 - p_a$. Successfully completing task $a$ will provide a known revenue $r_a$. Unsuccessfully completing any task will generate no revenue, we label this $r_\emptyset = 0$. Successful completing task $a$ will take time $T^s_a$ time units. Unsuccessfully completing task $a$ will take time $T^f_a$ time units. In the context of our motivating problem a success represents the purchase of an ad request by an in-network advertiser, and a failure represents the rejection of an ad request by an in-network advertiser.

$T^f_a, T^s_a$ are random variables with known means $\mu^f_a, \mu^s_a$ and known standard deviations $\sigma^f_a$ and $\sigma^s_a$. We must produce a policy in the form of a sequence $\pi$ in which to attempt tasks. Tasks in the sequence will be attempted in order until the first successful tasks.

A statically optimal policy is a sequencing of the tasks such that the expected revenue for attempting tasks in this sequence is maximal. In other words, an optimal policy is a permutation $\pi = P(1, N)$ such that $\pi(i)$ indicates the order in which task $i$ is to be attempted; this permutation is chosen at time $t = 0$ and remains fixed over time.

Equations 2.1 and 2.2 define a well behaved penalty function using common RM literature, see for example Gallago and van Ryzin [19]. These assumptions make reasoning about our problem easier and allow us to develop solution methods with a bounded optimality gap, we
discuss these in section 5.1. These equations are an instance of the "non-constant penalty for exceeding the deadline" family of penalty functions we discussed in section 1.1.

An alternative penalty function which belongs to the family of "constant penalty for exceeding the deadline" functions can be parameterized over both the elapsed time units, and the last solicited advertiser. Such a function gives the penalty associated with receiving a response from the last solicited advertiser \( a \) solicited after \( t \) time units have elapsed since the first advertiser in our sequence was solicited. This penalty, \( c\left((t - t^{max})^+, a\right) \) will be 0 until \( t > t^{max} \) (the contract deadline has passed).

We use the definition given below, it is convenient as it represent the situation that net-revenue is non-negative (ad networks do not pay fees for missing deadlines), however they make no profit on responses that are not filled in time. While our approach works for any monotonically increasing penalty function we use the penalty function in equation 2.3 for our empirical evaluations in section 12.1.

\[
c\left((t - t^{max})^+, a\right) = \begin{cases} 
0 & : t \leq t^{max} \\
r_a & : t > t^{max}
\end{cases} \tag{2.3}
\]

Please note that in the following sections we develop our solution techniques and optimality bounds based on the penalty function in equations 2.1 and 2.2, but that those solution techniques may be applied to penalty functions such as the one given in equation 2.3 (though in this case our solutions do not have a known optimality gap). In our empirical evaluation we present results given penalty functions from either family, in particular we use equations 2.1 and 2.2 as representative of the non-constant family, and 2.3 as representative of the constant family.
2.2 Problem Formulation

In this section we develop an objective function for the problem we’ve introduced. We assume that both the probability of success of a task, and the distributions describing the time spent in attempting it are independent of the probabilities of any other task. The validity of the assumption will be discussed at the end of section 2.3.

Define the following mutually independent random variables:

\[ X_i = 1 \text{ if success at task } i, 0 \text{ otherwise} \]
\[ T_{if} = \text{time spent to fail at task } i \]
\[ T_{is} = \text{time spent to succeed at task } i \]

Let \( \overline{X}_i = 1 - X_i \).

Notice that the following is true:

\[ p_i = \mathbb{E}[X_i] \quad \text{probability of success at task } i \]
\[ \mu_{if} = \mathbb{E}[T_{if}] \quad \text{mean failure time of task } i \]
\[ \mu_{is} = \mathbb{E}[T_{is}] \quad \text{mean success time of task } i \] (2.5)

For a given permutation/policy \( \pi \), \( \pi(i) \) is the index of the task that is attempted \( i \)’th. Let \( \rho(\pi) \) represent the net revenue from attempting a particular permutation \( \pi \). \( \rho(\pi) \) is defined as a sum of \( N \) terms \( \psi_{\pi(i)} \):
\[ \rho(\pi) = \sum_{i=1}^{N} \left( \psi_{\pi(i)} \right) \]  

(2.6)

Each term \( \psi_{\pi(k)} \) is defined:

\[ \psi_k = X_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( X_{\pi(i)} \right) \cdot \left( r_{\pi(k)} - c^1 \left( t^{max}, T^{s}_{\pi(k)} + \sum_{j=1}^{k-1} T^{f}_{\pi(j)} \right) \right) \]  

(2.7)

By independence and linearity of expectation the expected net revenue is:

\[ \mathbb{E}[\psi_{\pi(k)}] = \mathbb{E}[X_{\pi(k)}] \cdot \prod_{i=1}^{k-1} \left( \mathbb{E}[X_i] \right) \cdot \left( r_{\pi(k)} - \mathbb{E} \left[ c^1 \left( t^{max}, T^{s}_{\pi(k)} + \sum_{j=1}^{k-1} T^{f}_{\pi(j)} \right) \right] \right) \]  

(2.8)

After applying the relabellings from equation 2.5

\[ \mathbb{E}[\psi_{\pi(k)}] = p_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( 1 - p_{\pi(i)} \right) \cdot \left( r_{\pi(k)} - \mathbb{E} \left[ c^1 \left( t^{max}, T^{s}_{\pi(k)} + \sum_{j=1}^{k-1} T^{f}_{\pi(j)} \right) \right] \right) \]  

(2.9)

Thus, the expected net revenue of a permutation/policy \( \mathbb{E}[\rho(\pi)] \) is:

\[ \mathbb{E}[\rho(\pi)] = \mathbb{E} \left[ \sum_{k=1}^{N} \left( \psi_{\pi(k)} \right) \right] \]

\[ = \sum_{k=1}^{N} \left( p_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( 1 - p_{\pi(i)} \right) \cdot \left( r_{\pi(k)} - \mathbb{E} \left[ c^1 \left( t^{max}, T^{s}_{\pi(k)} + \sum_{j=1}^{k-1} T^{f}_{\pi(j)} \right) \right] \right) \right) \]  

(2.10)

Expected net revenue is the optimization objective for our problem.
2.3 Static Optimization Formulation

In this section we define some constraints needed to make the problem computationally tractable, and we formulate the static optimization problem in terms of those constraints. As the formulation given in this section is not in standard form standard convex optimization techniques do not apply to it in an obvious way. In section 4.1 we reformulate the optimization problem into a form in which the dynamic programming technique can be successfully applied.

To make our problem computationally tractable we define certain auxiliary variables to help introduce bounds on our computation. As we treat time as discrete variable we must set reasonable computational bounds over the domain of this variable. We define $t_{a_{\text{stop}}}$ for $a \in \mathcal{A}$ where $r_a - c_1(t_{\text{max}}, t_{a_{\text{stop}}}) = 0$; that is, each task $a$ has a time value $t_{a_{\text{stop}}}$ after which it will no longer be profitable to attempt task $a$ (even if we were guaranteed a success). Let’s define $t_{\text{stop}} = \max(t_{a_{\text{stop}}} \mid a \in \mathcal{A})$. $t_{\text{stop}}$ represents the number of time units past $t_{\text{max}}$ after which it will always be unprofitable to continue attempting tasks in set $\mathcal{A}$. The $t_{\text{max}}$ deadline for the set of tasks should be no more than $t_{\text{stop}}$ time units.

We illustrate $t_{\text{stop}}$ with an example:

- Suppose that $t_{\text{max}} = 10$, and that the revenue for advertiser $i$ is $r_i = 25$; notice that after $t = 15$ we it would never be profitable to solicit advertiser $i$ regardless of the probability that we will succeed.

- We call the maximum such stopping value over our full collection of advertisers $t_{\text{stop}}$.

Figure 2.1 presents a graphical illustration of this example.
It seems clear that the cost function $c^1$ penalizing solution lateness would have the property $\max(r_1, ..., r_N) \leq c^1(t^{\max}, t^{\text{stop}})$. That is, the penalty associated with this lateness value is no less than the greatest possible revenue from any of the tasks in the task set. Notice that under our definition $r_a - c^1(t^{\max}, t^{\text{stop}}) \leq 0 \ \forall \ a \in A$. We have defined $c^1$ to be a convex function with domain $\mathbb{Z}_{\geq 0}$ and with the value $c^1(0, t^{\max}) = 0$ (assuming that $t^{\max} > 0$); thus $c^1$ is a monotonically non-decreasing function. As $c^1$ is a monotonically non-decreasing it must be true that $\max(r_1, ..., r_N) \leq c^1(t^{\max}, t^{\text{stop}})$ under our definition of $t^{\text{stop}}$. Attempting tasks at or after time $t^{\text{stop}}$ is guaranteed to be unprofitable.

We begin by defining our objective function $f(\pi)$ developed in section 2.2 and factoring in the variables introduced to limit computation, the expected value of a policy $\pi$:

$$f(\pi) = \mathbb{E}(\rho(\pi)) = \sum_{i}^{N} p_{\pi(i)} \cdot \prod_{j<i}(1 - p_{\pi(j)}) \cdot \max\{\{t^{\pi(j)\text{stop}} | j>i\}\} \sum_{t=0}^{\max\{t^{\pi(j)\text{stop}} | j>i\}} \left((r_{\pi(i)} - c^1(t^{\max}, t)) \cdot Pr(T^{s}_{\pi(i)} + \sum_{\pi(j)<\pi(i)} T^f_{\pi(j)}) = t)\right)$$

(2.11)
Next we introduce the constraints and give the full static optimization problem (parameterized on set $\mathcal{A}$, and $t^{max}$) we label this $P(\mathcal{A}, t^{max})$:

$$\max_{\pi(1) \ldots \pi(N)} f(\pi)$$

Subject To:

- $y_{i,j} \in \{0, 1\}$ \hspace{1cm} $\forall (i, j) \in \{1, \ldots, N\} \times \{1, \ldots, N\}$
- $\pi(i) \in \mathbb{Z}_{\geq 0}$ \hspace{1cm} $\forall i \in \{1, \ldots, N\}$
- $\pi(i) \leq N$ \hspace{1cm} $\forall i \in \{1, \ldots, N\}$
- $\pi(j) = \sum_i i \cdot y_{i,j}$ \hspace{1cm} $\forall j \in \{1, \ldots, N\}$
- $\sum_i y_{i,j} = 1$ \hspace{1cm} $\forall i \in \{1, \ldots, N\}$
- $\sum_j y_{i,j} = 1$ \hspace{1cm} $\forall j \in \{1, \ldots, N\}$

In optimization 2.12 above the variables $y_{i,j}$ represent the scheduling of task $i$ to be the $j^{th}$ attempted task. Thus the constraint set $\forall i \in \{1, \ldots, N\}$ ($\sum_j y_{i,j} = 1$) represents the constraint that each task is assigned to be attempted in only one position in the order. The constraint set $\forall j \in \{1, \ldots, N\}$ ($\sum_i y_{i,j} = 1$) represents the constraint that each position in the order of attempted tasks has exactly one assigned task.

At this point we would like to note that for the ad ordering application described in previous sections the independence assumption is unlikely to be true. Figure 1.2 illustrates the fact that different advertisers will purchase web user and market segment information from one of several “Ad Data Marketplaces” these marketplaces in turn share data with each other. The nature of the data sharing between Ad Data Marketplaces is neither publicly available nor known to be deterministic. Clearly under these circumstances decisions made by disparate advertisers (though based on potentially similar information) will not be independent. How-
ever the complexities of this situation make some simplifying assumption necessary and this is why we choose to model these decisions as independent.

### 2.4 Expectation Relaxation for Stochastic Variables

In this section we present a relaxation of our original problem, which we label $\hat{P}(A, t^{\text{max}})$. This solution to the relaxed problem gives an upper bound on the expected net revenue for the original problem. The relaxation entails replacing each random variable in the problem formulation with it’s expected value, that is, for each $i$, $T_i^f$ and $T_i^s$ will be replaced with $\mu_i^f$ and $\mu_i^s$ respectively. We again use our definitions from section 2.2.

**Theorem 2.1.** The solution to the relaxed problem resulting when all terms $T_i^f$ and $T_i^s$ in the objective of our original problem (as described in section 2.2) are substituted with $\mu_i^f$ and $\mu_i^s$ respectively is an upper bound solution to our original problem.

The proof of this theorem is provided in Appendix A.

### 2.5 A Second Relaxation

In this section we relax the previous relaxation from section 2.4 to get a second order relaxation which (we label $\hat{\hat{P}}(A, t^{\text{max}})$) the solution to which serves as an upper bound for both the relaxed problem from section 2.4, and our original problem.

This second relaxation entails two steps. First we replace each term $\mu_i^f$, and each term $\mu_i^s$ in the objective function with the term $\min(\mu_i^f, \mu_i^s)$. Next, we ensure that each term $r_i - c^1(...) = r_i - c^1(...)$ in our objective is non-negative by substituting each such term with the term
max(r_i - c^1(...), 0). In the equation below we make use of the relaxed objective function in equation A.4:

**Theorem 2.2.** The solution to the relaxed problem resulting when all terms $\mu_i^f$ and $\mu_i^s$ in the objective of our previously relaxed problem (as described in section 2.4) are substituted with $\min(\mu_i^f, \mu_i^s)$, and each term $r_i - c^1(...)$ is substituted with $\max(r_i - c^1(...), 0)$ is an upper bound solution to our original problem.

The Proof of this theorem is provided in Appendix B.
Chapter 3

Heuristic Solution to the Static Problem

3.1 Solving the Static Problem Heuristically

A brute force solution to this problem is intractable for even small numbers of advertisers. The problem seems well suited to Feo and Resende and Ribeiro’s Greedy Randomized Adaptive Search Procedure (GRASP) [34] meta-heuristic. We chose the GRASP metaheuristic because it makes few assumptions about the problem structure, and does not necessitate careful consideration of primitives specific to the metaheuristic framework (such as crossover in genetic algorithms). GRASP is also primitively parallelizable, so it can be easily scaled up to make full use of all available parallel computing power. Whereas when parallelizing a genetic algorithm implementation network topologies and solution recombination methods can significantly affect the value of the resulting solution [3]; a parallel GRASP implementation should not be affected by these attributes.
We implemented an instance of this meta-heuristic for the purpose of computing a static sequence/permutation in which advertisers will be solicited. Once the solution is calculated advertisers are solicited in the order indicated in the solution until a time at which no advertiser can be profitably solicited. This time is calculated exactly as the value $t^{stop}$ from section 2.3 though the set of available actions over which this value is calculated may be reduced after each solicitation. We first describe the algorithm at a high level, next algorithm 1 gives the pseudo-code for our local search, finally we present a description of subroutines used in our pseudo-code.

1. Our local search begins by shuffling the advertisers into a random permutation.

2. The incumbent solution (the current best) generates a fixed window (say 1000) of successor solutions by one of two methods.

3. Method 1 <Intensify>: A child solution is a clone of the parent, but the order of 2 advertisers (chosen at random) is swapped (known as 2-swap).

4. Method 2 <Diversify>: The sub-sequence between 2 points in the permutation (chosen at random) is shuffled (known as scramble).

5. When the improvement from one window to the next increases(decreases), a temperature term increases(decreases) proportionate to the window to window change.

6. A higher temperature makes method 1 (intensify) more likely, a lower temperature makes method 2 (diversify) more likely.

7. When the window to window improvement becomes lower than a threshold execution will terminate.

In algorithm 1 we make use of several subroutines which we did not define. Below we will briefly describe these.
Algorithm 1 Grasp_obos

1: procedure Grasp_obos(\(\mathcal{A}\))  \quad \triangleright \text{Set } \mathcal{A} \text{ of advertisers.}
2: \quad x \leftarrow \text{PermuteAtRandom}(\mathcal{A})
3: \quad v \leftarrow \text{EvaluatePermutation}(x)
4: \quad v^- \leftarrow v - \epsilon
5: \quad \tau \leftarrow 1.0  \quad \triangleright \tau \text{ represents temperature.}
6: \quad \text{while } (v - v^- < \epsilon) \text{ do} \quad \triangleright \text{For some small } \epsilon.
7: \quad \quad v^- \leftarrow v
8: \quad \quad \text{for } i \in (0, ..., W) \text{ do} \quad \triangleright W \text{ is a window size.}
9: \quad \quad \quad \text{if } \text{GetRandom}() < \tau \text{ then}
10: \quad \quad \quad \quad x' \leftarrow \text{TwoSwap}(x) \quad \triangleright \text{Intensify.}
11: \quad \quad \quad \text{else}
12: \quad \quad \quad \quad x' \leftarrow \text{TwoShuffle}(x) \quad \triangleright \text{Diversify.}
13: \quad \quad \quad \text{end if}
14: \quad \quad \quad v' \leftarrow \text{EvaluatePermutation}(x')
15: \quad \quad \quad \text{if } v' > v \text{ then}
16: \quad \quad \quad \quad x \leftarrow x'
17: \quad \quad \quad \quad v \leftarrow v'
18: \quad \quad \quad \text{end if}
19: \quad \quad \text{end for}
20: \quad \quad \tau \leftarrow \tau \cdot \frac{v - v^-}{v^-}
21: \quad \text{end while}
22: \quad \text{return } x
23: \text{end procedure}

- **PermuteAtRandom**: Will permute the input set of advertisers into a random order.

- **EvaluatePermutation**: Returns the expected net revenue of the permutation according to equation 2.11.

- **GetRandom**: Returns a random value in (0, 1).

- **TwoSwap**: Returns a modified copy of the passed in sequence after an application of the classic 2-swap heuristic (see for example [28]).

- **TwoShuffle**: Returns a modified copy of the passed in sequence after an application of the classic scramble primitive (see for example [4]).
In our experiments our local search solution performs roughly 1% worse than the optimal solution on problem instances small enough that the optimal solution could be calculated reasonably quickly.
Chapter 4

Dynamic Problem Formulation

4.1 Markov Decision Process and Dynamic Program Formulation

We now formulate our original (stochastic) problem as a Markov decision process and give additional detail necessary to solve this problem as a dynamic program under a discrete time domain assumption. Note that classically such a formulation would include a discount factor (commonly indicated by $\gamma$) that would be applied to calculate the present value of future states. However, because the time horizon in our problems is discrete that term is dropped in the formulation.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
</table>

34
A state is described as a vector of no more than 3 elements. $s = (w, t, Q)$, where $w \in \emptyset \cup \{1, ..., N\}$, $t \in \{0, ..., \max(t_{q_{\text{stop}}}^{\text{stop}} | q \in Q)\}$ and $Q \subset \emptyset \cup \{1, ..., N\}$. Here, $w$ and $t$ are scalar variables, integer values but $Q$ is a vector variable. The scalar value $w$ represents either the index of the successfully completed task, or $\emptyset$ if no task has yet been successfully completed. The scalar value $t$ represents the amount of time units elapsed since the process of attempting mutually exclusive tasks began. Notice that when $t > t_{\text{max}}$, then $t - t_{\text{max}}$ represents how many time units have been expended since the deadline $t_{\text{max}}$ has passed. The set $Q$ represents the remaining set of tasks, or $\emptyset$ if there are no remaining tasks at state $s$.

<p>| State     | A state is described as a vector of no more than 3 elements. $s = (w, t, Q)$, where $w \in \emptyset \cup {1, ..., N}$, $t \in {0, ..., \max(t_{q_{\text{stop}}}^{\text{stop}} | q \in Q)}$ and $Q \subset \emptyset \cup {1, ..., N}$. Here, $w$ and $t$ are scalar variables, integer values but $Q$ is a vector variable. The scalar value $w$ represents either the index of the successfully completed task, or $\emptyset$ if no task has yet been successfully completed. The scalar value $t$ represents the amount of time units elapsed since the process of attempting mutually exclusive tasks began. Notice that when $t &gt; t_{\text{max}}$, then $t - t_{\text{max}}$ represents how many time units have been expended since the deadline $t_{\text{max}}$ has passed. The set $Q$ represents the remaining set of tasks, or $\emptyset$ if there are no remaining tasks at state $s$. |
|-----------|-------------------------------------------------------------------------------------------------|
| Initial   | In this problem our initial state set is a singleton; $I = {(\emptyset, t_{\text{max}}, \mathcal{A})}$ |
| State     |                                                                                                     |
| Final     | A state $s = (w, t, Q)$ is final ($s \in \mathcal{F}$) if either or both of these conditions hold:  |
| State     | 1. $Q = \emptyset$; that is, all available, potentially profitable tasks have already been attempted. |
| Set       | 2. $w \neq \emptyset$; that is, one of the attempted tasks has been successfully completed.            |
| Note      | Note that states where $t = \max(t_{q_{\text{stop}}}^{\text{stop}} | q \in Q)$ are subsumed by item |</p>
<table>
<thead>
<tr>
<th>State Space</th>
<th>((t^{max} + t^{stop}) \cdot (N + 1) \cdot 2^N) is a loose upper bound on the cardinality of the state space.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision</td>
<td>The decision (d) taken at state (s = (w, t, Q)) indicates the next task to be attempted. The decision must be chosen from the feasible, “potentially profitable&quot; tasks; so ((d \in \mathcal{Q})).</td>
</tr>
<tr>
<td>Feasible “potentially profitable” Decision</td>
<td>The feasible “potentially profitable” actions at state (s = (w, t, Q)) are all “potentially profitable” tasks not already attempted, that is tasks in the set (\mathcal{Q}).</td>
</tr>
</tbody>
</table>
| Transition Function | The transition function \(S' = S^M(s, d)\) gives the set \(S'\) of successor states of parent state \(s = (w, t, Q)\) when applying decision \(d\) (that is choosing \(d\) as the next task to attempt). At each transition we define \(\mathcal{U}\), a subset of \(\mathcal{Q}\) as follows: \(\{q \in \mathcal{U} \mid C(s, q) < 0 \land q \in \mathcal{Q}\}\), where \(C(s, q)\) refers to the contribution function which is defined below. The successor states are partitioned into success states \(\mathcal{K}_s\) and failure states \(\mathcal{K}_f\). \[
\mathcal{K}_s = \{(d, t', \mathcal{Q} - \mathcal{U} - \{d\}) \mid t < t' \leq \max\{t^{q^{stop}} \mid q \in \mathcal{Q}\}\} \\
\mathcal{K}_f = \{(\emptyset, t', \mathcal{Q} - \mathcal{U} - \{d\}) \mid t < t' \leq \max\{t^{q^{stop}} \mid q \in \mathcal{Q}\}\} \\
S' = \mathcal{K}_s \cup \mathcal{K}_f
\] |
<table>
<thead>
<tr>
<th>Decision Objective</th>
<th>We define the contribution function $C(s, d)$ for applying transition $S^M(s, d)$ to state $s = (w, t, Q)$, where $d \in Q$.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$C(s, d) = p_d \left( r_d - \mathbb{E} \left( c^1(t^{\text{max}}<em>d, t + T^d_s) \right) + \max(t^{\text{stop}}</em>{q</td>
</tr>
<tr>
<td></td>
<td>$(1 - p_d) \sum_{t' = 0}^{t^{\text{stop}}<em>{q' \in \varnothing \setminus {d}}} \sum</em>{q \in \varnothing \setminus {d}} C((w, t', Q - {d}, q))$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path Objective</th>
<th>Let us define a binary $\parallel$ operation ($\parallel : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$) thus:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$h \parallel k = \begin{cases} h : h \neq 0 \ k : \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td></td>
<td>Let $x_1 \parallel x_2 \parallel \ldots \parallel x_N$ represent the reduction $x_1 \parallel x_2 \parallel \ldots \parallel x_N$.</td>
</tr>
<tr>
<td></td>
<td>Note that this operation is associative. Then the path objective is:</td>
</tr>
<tr>
<td></td>
<td>$\parallel_{s \in S, q \in \varnothing} C(s, q) + f(s_f)$</td>
</tr>
<tr>
<td></td>
<td>Where $f(s_f)$ is the final value function defined below, and as before $s = (w, t, \varnothing)$</td>
</tr>
</tbody>
</table>

| Final Value Function | For $s \in \mathcal{F}$, where $s = (w, t, \varnothing)$. We use the symbol $s_f$ to indicate a state $s_f \in \mathcal{F}$. $r_w - c^1(t^{\text{max}}_s, t)$, Note that $r_\emptyset = 0$. |
The Bellman equation can be written:

\[ V(s) = \max_{q \in Q} C(s, q) + \sum_{s' \in SM(s, q)} V(s') \] (4.1)

where as before \( s = (w, t, \mathcal{Q}) \).

The state transitions can be visualized as traversing a state grid from left to right. Suppose that the horizontal axis of the grid represents time \( t \), and the vertical axis represents the set \( \mathcal{Q} \) produced as a result of an action (task attempt). Then tasks attempts chosen endogenously drive the current state up or down along the vertical axis, and exogenous delay times will monotonically drive the current state some distance to the left until a terminating state is encountered.

Figure 4.1 gives an example of this visualization for the case where \( \mathcal{I} = \{(\emptyset, 0, \{1, 2, 3, 4, 5\})\} \) and the decision variable has been set to \( d = 5 \), that is task 5 has been chosen to be explored next. The dashed edges represent the possible destination states after the action is taken. Note that we assume in this example that at least one time unit will be expended by any action.
\( Q = \{1, 2, 3, 4, 5\} \)

\( Q = \{1, 2, 3, 4\} \)

\( Q = \{1, 2, 3\} \)

\( t = 0 \quad t = 1 \quad t = 2 \quad t = 3 \quad \cdots \)

---

Figure 4.1: State Transitions
Chapter 5

Constructing a Policy and Online DP

5.1 Policy Determination

In the terminology in [33] the policy used in our solution method is a look-ahead policy implementing a lookup table approximation strategy which is pre-calculated in a tree search. The problem we describe is a instance of the more general problem of formulating an on-line optimization which would continuously optimize the order of tasks to be attempted while adjusting both the strategy of the optimization and the statistics upon which it is based as new information is obtained. In our problem we do not modify the policy once calculated, nor account for updates to statistics. For example we do not modify the value $T_a^f$ after unsuccessfully attempting task $a$ and learning the delay time of that unsuccessfully attempt. In calculating the lookup table necessary for our policy choice we employ the iterated Gauss-Seidel variation of the value iteration algorithm for finite state optimization (see for example [33], page 69). The proof of convergence at optimality of the value iteration algorithm is given in [33], pages 89-93.
In our implementation we begin by evaluating both the optimal value of the initial state and the action that must be taken at the initial state so as to realize that optimal value; this is done in the beginning of algorithm 3. The value iteration algorithm given in 2 recursively calculates the optimal value at each successor state of the state passed to it as an argument then chooses the successor state with the maximal such value and the action necessary to arrive at that successor state. The recursion in this calculation is terminated upon the detection of a final state (as described in table 4.1) the detection of these final states can be seen in the first two conditional statements of algorithm 2.

Below we give two algorithms which implement the solution method described in the preceding paragraph. Algorithm 2 implements value-iteration to produce a function $V$ which will give the optimal decision for any achievable state $s$. Algorithm 3 then produces this function and iteratively executes the policy that it encodes. Notice that as all achievable states are encoded in the function returned by algorithm 2 exogenous information encountered in the course of executing a given decision are accounted for in the function prior to their occurrence; this feature classifies our solution method as a look-ahead policy.

A detail that may be obscured by the psudo-code above is that the value table $V$ (calculated when executing algorithm 2) is itself the third argument of all triplets found in the co-domain of $V$. A naive implementation might use $O(t_{max} \cdot 2^N)$ space to store $O(t_{max} \cdot 2^N)$ such tables. However, if the third argument of each triplet in the co-domain instead stores a memory reference to a dynamically reallocated section of memory we will require only a single table with at most $O(t_{max} \cdot 2^N)$ cells. In other words, implementing this function with a “pass by reference” evaluation strategy exponentially reduces the space requirement by a factor of the maximal size of the table ($t_{max} \cdot 2^N$). In “pass by reference” the addresses of machine memory where argument data is stored are exchanged between the calling procedure and the called procedure; this strategy avoids the data duplication that occurs when data (other than a memory address) is replicated at each procedure call.
Algorithm 2 ValueIteration

1: procedure ValueIteration(s, V)
2:   (w, t, Q) ← s  \hspace{1em} \triangleright s \text{ decomposes into its components as in table 4.1.}
3:   if V(s) \neq (\emptyset, \emptyset, \emptyset) then \hspace{1em} \triangleright \text{Previously computed.}
4:     return V(s)
5:   else if |Q| = 0 \land w \neq \emptyset then \hspace{1em} \triangleright \text{Final State.}
6:     return (r_w - c^1(t_{\text{max}}, t, t, V)
7:   else
8:     \begin{align*}
9:       X^v &\leftarrow -\infty \\
10:      X^a &\leftarrow \emptyset \\
11:      \text{for } q \in Q \text{ do} & \\
12:       \begin{align*}
13:         v^s &\leftarrow p_q(r_q - \mathbb{E}[c^1(t_{\text{max}}, t + T_q)]) \\
14:         v^f &\leftarrow 0 \\
15:         \text{for } t' \in (t, \ldots, \max(t_{\text{stop}}^q | q \in Q)) \text{ do} & \\
16:           s' &\leftarrow (w, t', Q - \{q\}) \\
17:           (v', q', V) &\leftarrow \text{ValueIteration}(s', V) \hspace{1em} \triangleright V \text{ is updated.}
18:           v^f &\leftarrow v^f + \Pr(T_q^f = t' - t) \cdot v'
19:         \end{align*}
20:       \text{end for} \\
21:       \begin{align*}
22:         \text{if } X^v < v^s + (1 - p_q) \cdot v^f \text{ then} & \\
23:           X^v &\leftarrow v^s + (1 - p_q) \cdot v^f \\
24:           X^a &\leftarrow q
25:       \text{end if}
26:     \end{align*}
27:   \text{end for}
28:\end{align*}
29:   V(s) &\leftarrow (X^v, X^a, V) \hspace{1em} \triangleright V \text{ stores both the policy and its value when applied to } s.
30:   return V(s)
31: end procedure

5.2 Dynamic Programming Online Solution

In the process of solving the dynamic program for the static sequencing problem described above a table of sub-problems is calculated. It is notable that when retaining the solutions to these sub-problems we can optimally solve the online variation of this problem. In the online variation of this problem rather than computing an ordering we use the look-ahead policy encoded in the table generated during the solution of the static problem by algorithm 2. This table will return the optimal action to take from any reachable state thus it can be
Algorithm 3 ExecutingTheOptimalOrder

1: procedure OptimizeOrder(\(\mathcal{A}, t^{\text{max}}\))
2: \(V(k) \leftarrow (\emptyset, \emptyset, \emptyset) \ \forall k \in S\) \quad \triangleright \text{Initialize mapping } V.
3: \(\mathcal{Q} \leftarrow \mathcal{A}\)
4: \(\mathcal{G} \leftarrow (\emptyset, t^{\text{max}}, \mathcal{Q})\)
5: \((v, a, V) \leftarrow \text{ValueIteration}(\mathcal{G}, V)\) \quad \triangleright \text{After this all actions for all future states are precalculated.}
6: \(s \leftarrow \text{Attempt}(a)\)
7: \((w, t, \mathcal{Q}) \leftarrow s\) \quad \triangleright \text{s decomposes into its components as in table 4.1.}
8: while \(w = \emptyset \land |\mathcal{Q}| > 0\) do
9: \((v, a, V) \leftarrow V(s)\) \quad \triangleright \text{Update to } V \text{ here is not useful.}
10: \(s \leftarrow \text{Attempt}(s, a)\)
11: end while
12: return \(v\)
13: end procedure

used in the online version of the problem to choose the optimal action from any state that actually is encountered in the course of the executing a sequence of task attempts.

As tasks are attempted online each test will return an indication of success or failure and a positive integer number to indicate the amount of time units that the attempt took. We have pre-calculated optimal decisions for every set of tasks and every time \(t\) at which at least one of these tasks is “potentially profitable” when calculating the static solution. Thus at the end of the calculation we should have tabulated the optimal decision after any unsuccessful attempt. Intuitively, as we have calculated the optimal decisions for all possible sets of un-attempted tasks and all possible, and “potentially profitable” time instants we will have calculated the optimal decision for any state that is probabilistically transitioned into during the process of sequentially attempting tasks.

The principal difference between the static and online problems is that those variables that are unknown in the static variation of the problem (and thus the state space of all feasible “potentially profitable” successor states is examined) become known as the task attempts are actually made. While in the static variation of the problem table-construction and retrieval
occur simultaneously, in the online variation there are two clear phases. In the first phase the table is constructed as described thus far, this phase is potentially time intensive; we refer to this phase as the “construction phase”. In the second phase tasks are being attempted and the optimal actions to attempt are taken based on values calculated during the “construction phase”, we call this section phase the “in-use” phase. The second phase is time sensitive, as overly long delays at this phase may the state from which the optimal action is chosen, thus making the choice suboptimal. When the table calculated during the “construction phase” is small enough to fit into memory, it can be queried into in $O(1)$ time by using a hash-table data structure.
Chapter 6

Approximating the DP

6.1 A Scalable Approximation Strategy

Algorithm 2 is an $O(2^N)$ algorithm and is intractable for large values of $N$. In this section we develop a tractable algorithm for approximating the optimal policy. We use the relaxation presented in section 2.4 as a sub-problem in our design of a real-time dynamic programming (RTDP) (due to Barto et al [7]) approximation strategy.

Section 7.1 presents an algorithm for calculating an upper bound solution for our problem. The algorithm is pseudo-polynomial in $t_{stop}$, that is, it has complexity $O(t_{stop} \cdot N)$.

In algorithm 4 we apply the real time dynamic programming technique to our problem domain. The code is a straight-forward application of the pseudo-code from page 127 of [33]. This technique allows us to execute significantly fewer than $O(2^N)$ operations, in-fact the algorithm is $O(t_{stop} \cdot M \cdot N^2)$ where $M$ is determined by the stopping criterion (assuming that the criteria is a fixed number of iterations). The call to SimulateAttempt represents
choosing a sample from the Bernoulli random variable representing the success or failure of an attempt, and a sample from the distribution representing the delay time for the attempt.

In algorithm 5 we apply the table pre-calculated in algorithm 4 to a dynamic problem. Algorithm 5 is nearly identical to algorithm 3 with one important exception; $\tilde{V}$ is a specially defined look-up into the table rather than a direct look-up. Let $\tilde{V}(s)$ find the entry of the “nearest neighbor” of state $s$ in the table and return its value. Implementing a nearest neighbor function in this context is non-trivial, we discuss its design in section 9.1. A fast “nearest neighbor” function is only necessary in the dynamic context as we must find a feasible “potentially profitable” action to execute quickly. In a static context (calculating a static order) where more time is available to execute the query we may instead calculate $\tilde{V}(s)$ directly by executing $V = RTDP(\mathcal{Q}, t)$ and then $V(s)$ where as before $s = (w, t, \mathcal{Q})$.

---

**Algorithm 4** RTDP

1: procedure RTDP($\mathcal{A}, t^{\text{max}}$)  
2: $V(k) \leftarrow (\emptyset, \emptyset) \forall k \in \mathcal{S}$ \texttt{\textcopyright} Initialize mapping $V$.  
3: while Termination Criterion do  
4: \hspace{1em} $\mathcal{Q} \leftarrow \mathcal{A}$  
5: \hspace{1em} $t \leftarrow 0$  
6: \hspace{1em} $w \leftarrow \emptyset$  
7: \hspace{2em} while $|\mathcal{Q}| > 0 \land w = \emptyset$ do  
8: \hspace{3em} $s \leftarrow (w, t, \mathcal{Q})$ \texttt{\textcopyright} declare state $s$.  
9: \hspace{3em} $X^v \leftarrow -\infty$  
10: \hspace{3em} $X^a \leftarrow \emptyset$  
11: \hspace{4em} for $q \in \mathcal{Q}$ do  
12: \hspace{5em} $s' \leftarrow (\emptyset, \max(0, t + \mu_q^f), \mathcal{Q} - \{q\})$ \texttt{\textcopyright} declare state $s'$.  
13: \hspace{5em} if $V(s) = (\emptyset, \emptyset)$ then  
14: \hspace{6em} $(Y^v, Y^a) \leftarrow \text{UpperBound}(s', t^{\text{max}})$  
15: \hspace{6em} else  
16: \hspace{7em} $(Y^v, Y^a) \leftarrow V(s')$  
17: \hspace{4em} end if  
18: end for  
19: end while  
20: return $V$
Algorithm 4 RTDP (continued)

18: \[ Y^v \leftarrow Y^v + p_q \cdot r_q \]
19: \[ \text{if } Y^v > X^v \text{ then} \]
20: \[ X^v = Y^v \]
21: \[ X^a = Y^a \]
22: \[ \text{end if} \]
23: \[ \text{end for} \]
24: \[ \text{if } X^a = \emptyset \text{ then} \]
25: \[ Q \leftarrow \emptyset \]
26: \[ \text{else} \]
27: \[ V(t, Q) \leftarrow (X^v, X^a) \]
28: \[ s \leftarrow \text{SIMULATEATTEMPT}(s, X^a) \]
29: \[ (w, t, Q) \leftarrow s \]
30: \[ \text{end if} \]
31: \[ \text{end while} \]
32: \[ \text{return } (V(s), V) \]
33: \[ \text{end procedure} \]

Problem \( \hat{P}(A, t_{\text{max}}) \) (from section 2.5) has the optimal substructure property which allows it to be solved by dynamic programming. Our algorithm for solving problem \( \hat{P}(A, t_{\text{max}}) \) is a slight variation of the classic dynamic programming solution to the 0/1 knapsack problem due to Richard Bellman [8].

Algorithm 4 is a canonical implementation of the RTDP algorithm and is presented for clarity. It converges to optimality at termination, however this would be too costly in practice so our termination criterion is looser than the one given by Barto et al. In fact our implementation borrows heavily from Bonet et al [12, 11], McHahan et al [27], Besse et al [10], and most significantly from Smith and Simmons [37]. It would not prove helpful to the reader to reproduce our implementation of RTDP in full as it borrows from many of the refinements cited here, in particular from Smith and Simmons. In summary, our algorithm uses tightening upper and lower bound approximations to recover a solution that is within \( \epsilon \) of the optimal solution (for a user specified \( \epsilon \)); and it does so in reasonable time. The lower bound valuation of a given state is calculated by our heuristic solution static solution.
Algorithm 5 ExecutingTheApproximateOrder

1: procedure OptimizeOrder($\mathcal{A}, t^{max}$)
2:   $\mathcal{Q} \leftarrow \mathcal{A}$
3:   $\mathcal{I} \leftarrow (\emptyset, t^{max}, \mathcal{Q})$
4:   ($v, a, V$) $\leftarrow$ RTDP($\mathcal{A}, t^{max}$) ▷ After this actions for many future states are precalculated.
5:   $s \leftarrow$ Attempt($a$)
6:   ($w, t, \mathcal{Q}$) $\leftarrow$ $s$ ▷ $s$ decomposes into its components as in table 4.1.
7:   while $w = \emptyset \land |\mathcal{Q}| > 0$ do
8:     ($v, a$) $\leftarrow$ $\tilde{V}(s)$
9:     $s \leftarrow$ Attempt($s, a$)
10:   end while
11: return $v$
12: end procedure

to the static problem (given in section 3.1). The upper bound valuation of a given state is calculated by an application of our relaxations to the problem and an application of Richard Bellman’s technique for the 0/1 Knapsack; the details are given in section 7.1. Finally, to bootstrap our RTDP algorithm we make use of our statically computed GRASP solution.
Chapter 7

Proving the Upper Bound

7.1 An Upper Bound

7.1.1 Preliminaries for Deriving the Upper Bound

In section 2.3 and table 4.1 we described how the introduction of a $t_{stop}$ term can limit the state space of the problem from an infinite state space ($t \in (0, ..., \infty)$) to a finite one ($t \in (0, ..., t_{stop})$). We make use of this reduced state space when writing a dynamic programming solution for this problem. Specifically, we will only consider sequences of tasks such that the execution of a sequence takes no more than $t_{stop}$ time units to conclude.

Formally, the above means that we will be solving problems of the form $\hat{P}(\mathfrak{l},x) \forall \mathfrak{l} \subseteq A, 0 \leq x \leq t_{max}$. This domain restriction reduces the number of such problems to a finite collection, however examining all subsets $\mathfrak{l} \subseteq A$ still requires solving $O(2^N \cdot t_{stop})$ sub-problems which is intractable for large $N$. 
We can in fact solve only $O(N \cdot t^{stop})$ sub-problems and still achieve optimality. We achieve this further state space reduction by limiting the number of subsets examined to only the set of all sub-sequences beginning at an arbitrary point in the sequence of tasks when ordered in non-increasing order by revenue, and terminating at the end of the ordered sequence.

With this further restriction the sub-problems we are solving are:

$$\hat{\mathcal{P}}(\{R(i), ..., R(N)\}, x) \forall i \in (1, ..., N), 0 \leq x \leq t^{\text{max}},$$

where $R(i)$ indicates the $i$’th task when tasks are ordered in non-increasing order by revenue, that is $R(i) \leq R(j) \implies r_i \leq r_j$. To be able to make this domain restriction we need to ensure that:

1. The optimal solution is a permutation in which tasks are non-increasing in revenue.

2. The optimal solution may be computed from sub-problems parameterized on tasks $\{R(i), ..., R(N)\}$.

Point 1 is a consequence of theorem 7.1, point 2 is an implication of the series of lemmas from section 7.1.7. Algorithm 6 shows how the upper bound may be calculated, and theorem 8.1 states that the calculation is correct. Algorithm 7 uses the standard solution reconstruction technique for dynamic programs to reconstruct the set of tasks in the optimal solution, ordering these into non-increasing order of revenue provides the optimal solution.

**Theorem 7.1.** An optimal solution to the relaxed problem from section 2.5 will always be ordered in non-increasing order of revenue.

**Proof.** This can be shown by contradiction: Suppose that the optimal solution is not non-increasing in the revenue of the tasks. By our assumption the optimal solution must contain a task sub-sequence $j, i$ where $r_i > r_j$. However, as revenue terms as well as probability terms are positive: $p_i \cdot r_i + (1 - p_i) \cdot r_j > p_j \cdot r_j + (1 - p_j) \cdot r_i$. Thus the solution was sub-optimal and this is a contradiction. \qed
Theorem 7.1 is possible as our relaxations from section 2.4 and 2.5 removed all uncertainty regarding how many time units are expended when attempting a given sequence of tasks. Thus the optimal sequence will non-increasing in revenue (of course which tasks participate in the optimal order is not exactly specified by this fact). We can deduce that if a task \( i \) is the first attempted task in the optimal order, then every subsequent task \( j \) attempted will have revenue \( r_j \leq r_i \).

### 7.1.2 Definitions, Assumptions, and Simplified Notation

We briefly restate the definitions used in this proof, state some necessary assumptions, and introduce some simplifying notation to reduce the size of our equations. In our definitions in section 7.1.3 we also introduce a \( N+1 \)'st task to explicitly represent the null action (no action).

### 7.1.3 Definitions

- \( t \) represents time since the start.
- \( \mathcal{A} = \{1,...,N\} \).
- \( \pi \in P(X) \mid X \subseteq \mathcal{A} \).
- \( g(t,N+1) \) represents the maximum expected revenue of the empty sequence \( \pi = () \),
  \[
g(t,N+1) = 0 \quad \forall t \in (0,\ldots,t_{\text{stop}})
\]
- \( \mu_{N+1} = 0, r^*[N+1] = 0. \)
- All solutions evaluated by \( g^* \) end by attempting task \( N+1 \).
7.1.4 Simplified Notation

For the purposes of our proofs in this section we made some simplifications to reduce the size of our equations.

\[ q_i = 1 - p_i \]
\[ \mu_i = \min(\mu_i^f, \mu_i^s) \]
\[ r^*[i] = r_i - c^1(t_{max}, t) \]

The notation \(\pi(x :)\) indicates the sub-sequence of \(\pi\) starting at index \(x\) and continuing through to index \(|\pi|\). Similarly, the notation \(\pi(: x)\) indicates the sub-sequence of \(\pi\) starting at index 0 of \(\pi\) and continuing through to index \(x\).

7.1.5 Assumptions

Some reasonable assumptions are necessary for our proof:

Assumption 1-\(\mathcal{A}\).

\[ \forall i \in \mathcal{A}, \]
\[ 0 < p_i < 1 \]
Assumption 2-\(\mathcal{A}\).

\[ \forall i \in \mathcal{A}, \forall j \in \mathcal{A}, \forall t \in \{0, ..., \infty\}, \]

\[ i \neq j \implies p_i \cdot r^*[t + \mu_i] \neq p_j \cdot r^*[t + \mu_j] \]

7.1.6 An Upper Bound Proof

The statement of equation B.1 in the simplified notation is:

\[
\hat{f} = \sum_{k=\pi(1)}^{\pi(|\pi|)} \left( p_{\pi(k)} \left( \prod_{i=\pi(1)}^{k-1} q_{\pi(k)} \right) \cdot R^*_{\pi(k)} \left[ \sum_{j=\pi(1)}^{k} \mu_{\pi(i)} \right] \right)
\]  \hspace{1cm} (7.1)

Define the auxiliary function \(\hat{f}^*\):

\[
\hat{f}^* = \begin{cases} 
\hat{f}^*(\pi) : & \sum_{i=\pi(1)}^{\pi(|\pi|)} \mu_{\pi(i)} \leq t_{\text{stop}} \\
-\infty : & \text{otherwise}
\end{cases}
\]  \hspace{1cm} (7.2)

Define the auxiliary function \(g\):

\[
g(t, \pi) = \begin{cases} 
p_{\pi(1)} \cdot r^*[t + \mu_{\pi(1)}] + q_{\pi(1)} \cdot g(t + \mu_{\pi(1)}, \pi(2 :)) : & |\pi| > 0 \\
0 : & |\pi| = 0 \land t \leq t_{\text{stop}} \\
-\infty : & |\pi| = 0 \land t > t_{\text{stop}}
\end{cases}
\]  \hspace{1cm} (7.3)
Define the auxiliary function $g^*$:

$$g^*(t, i) = \begin{cases} 
    p_i \cdot r^*[t + \mu_i] + q_i \cdot \max_{j \in \{i+1, \ldots, N+1\}} \left( g^*(t + \mu_i, j) \right) & : i < N + 1 \\
    0 & : i = N + 1 \land t \leq t^{stop} \\
    -\infty & : i = N + 1 \land t > t^{stop}
\end{cases} \quad (7.4)$$

$\forall x \in (0, \ldots, t^{stop}), \forall i \in \mathcal{A}$, $g^*(x, i) \to \mathbb{R}$ represents the highest expected revenue as evaluated by $\hat{f}$ of any sequence of attempts of unique tasks that begins with task $i$, subsequently only attempts tasks with revenue no more than $r_i$, and the full sequence of attempts takes no more than $x$ time units to complete. In the series of lemmas in section 7.1.7 we derive the relationship between $\hat{f}$ and $g^*$.

### 7.1.7 Lemmas

**Lemma 7.2.** $\max_{\pi} \left( \hat{f}(\pi) \right) = \max_{\pi} \hat{f}^*(\pi)$

*Proof.* $\emptyset \subseteq \mathcal{A}$. If $\pi = ()$ then $\hat{f}(\pi) = \hat{f}^*(\pi) = 0$. Thus, $\max_{\pi} \left( \hat{f}(\pi) \right) \geq 0$ and $\max_{\pi} \hat{f}^*(\pi) \geq 0$. As the maximum of either function is non-negative the piece-wise definition of $\hat{f}^*$ indicates that $\max_{\pi} \left( \hat{f}(\pi) \right) = \max_{\pi} \hat{f}^*(\pi)$. \qed

**Lemma 7.3.**

$\forall \pi, \forall y \in \{1, \ldots, |\pi|\}$, $t = \sum_{x=\pi(1)}^{\pi(y)} \mu_{\pi(x)}$

$$\left( |\pi| > 0 \land \sum_{i=\pi(1)}^{\pi(|\pi|)} \mu_{\pi(i)} \leq t^{stop} \right) \implies \left( \hat{f}(\pi) = g(t, \pi) \right)$$
Proof. Suppose this is true for $\pi$ when $|\pi| = z$. For the case that $|\pi| = z + 1$, we rearrange the definition of $\hat{f}^*$:

$$\hat{f}^*(\pi) = p_{\pi(1)} \cdot r^*[\mu_{\pi(1)}] + q_{\pi(1)} \cdot \hat{f}^*(\pi(2 :))$$

By our assumption:

$$\hat{f}^*(\pi) = p_{\pi(1)} \cdot r^*[\mu_{\pi(1)}] + q_{\pi(1)} \cdot g(t, \pi(2 :))$$

for all $t$ that satisfy the assumption conditions.

But from the definition of $g$ that applies to our case:

$$g(t', \pi) = p_{\pi(1)} \cdot r^*[t' + \mu_{\pi(1)}] + q_{\pi(1)} \cdot g(t' + \mu_{\pi(1)}, \pi(2 :))$$

We know that the term $t = t' + \mu_{\pi(1)}$ satisfies the conditions on $t$ in our assumption for all values of $\pi(1)$. Thus:

$$\hat{f}(\pi) = g(t, \pi)$$

Lemma 7.4. $\forall \pi \hat{f}^*(\pi) = g(0, \pi)$

Proof.

- Suppose that $\sum_{i=\pi(1)}^{\pi(|\pi|)} \mu_{\pi(i)} \leq t^{stop}$, and $|\pi| = 0$. From its definition we know that $t^{stop} \geq 0$.

In this case it is clear from the definitions that $\hat{f}^*(\pi) = g(0, \pi) = 0$.

- Suppose that $\sum_{i=\pi(1)}^{\pi(|\pi|)} \mu_{\pi(i)} \leq t^{stop}$, and $|\pi| > 0$.

Rearranging equation 7.3:

$$\hat{f}^*(\pi) = p_{\pi(1)} \cdot r^*[\mu_{\pi(1)}] + q_{\pi(1)} \cdot \hat{f}^*(\pi(2 :))$$
From the definition of $g$:

$$g(0, \pi) = p_{\pi(1)} \cdot r^*[\mu_{\pi(1)}] + q_{\pi(1)} \cdot g(\mu_{\pi(1)}, \pi(2 :))$$

By lemma 7.3, and for this case:

$$g(\mu_{\pi(1)}, \pi(2 :)) = \hat{f}(\pi) = \hat{f}^*(\pi)$$

Thus:

$$\hat{f}^*(\pi) = g(0, \pi)$$

• Suppose that $\sum_{i=\pi(1)}^{\pi(|\pi|)} \mu_{\pi(i)} > t_{stop}$, and $|\pi| = 0$. It is clear from the definitions that $\hat{f}^*(\pi) = g(0, \pi) = -\infty$.

• Suppose that $\sum_{i=\pi(1)}^{\pi(|\pi|)} \mu_{\pi(i)} > t_{stop}$, and $|\pi| > 0$. It is clear from definition that $\hat{f}^*(\pi) = -\infty$.

Define $z$ such that:

$$\sum_{i=\pi(1)}^{\pi(z)} \mu_{\pi(i)} \leq t_{stop} < \sum_{i=\pi(1)}^{\pi(z+1)} \mu_{\pi(i)}$$

Rearranging equation 7.4:

$$g(0, \pi) = g\left(0, \pi(1 : z)\right) + g\left(\sum_{i=\pi(1)}^{\pi(z)} \mu_{\pi(i)}, \pi(z + 1 :)ight)$$

From the definitions of $g$ and $z$

$$g\left(\sum_{i=\pi(1)}^{\pi(z)} \mu_{\pi(i)}, \pi(z + 1 :)ight) = -\infty$$

thus:

$$g(0, \pi) = \hat{f}^*(\pi) = -\infty$$
Lemma 7.5.

$$\forall t \in \{0, \ldots, \infty\}, \forall i \in A$$

$$\max_{\pi} \left( g(t, \pi) \right) = \max_{i \in \{1, \ldots, N+1\}} \left( g^*(t, i) \right)$$

Proof. Let $$\pi^* = \max_{\pi} \left( g(t, \pi) \right)$$, and let $$i^* = \arg \max_{i \in \{1, \ldots, N+1\}} \left( g^*(t, i) \right)$$.

- Suppose that $$i^* = N + 1$$, and that $$t > t^{\text{stop}}$$; then:
  - If $$\pi^* = ()$$, then from the definitions of $$g$$ and $$g^*$$ it is clear that:
    $$g(t, \pi^*) = g^*(t, i^*) = -\infty$$
  - Otherwise, suppose that $$g(t, \pi^*) = g^*(t, i^*) = -\infty$$ when $$|\pi^*| \leq k$$ and $$t > t^{\text{stop}}$$. If $$|\pi^*| = k + 1$$ then by definition:
    $$g(t, \pi^*) = p_{\pi^*} \cdot r^*[t + \mu_{\pi^*}] + q_{\pi^*} \cdot g(t + \mu_{\pi^*}, \pi^* \langle 2 :) \rangle,$$
    $$|\pi^* \langle 2 :) \rangle| = k$$ and $$t + \mu_{\pi^*} > t > t^{\text{stop}}$$ as $$\mu_x \geq 0 \ \forall x \in A$$.
    Thus by our inductive assumption, assumption 1-$$A$$ and because:
    $$\forall x \in A, \forall y \in (0, \ldots, \infty) \ q_x > 0, p_x > 0, \infty > r^*[y] > 0$$
    $$g(t, \pi^*) = g^*(t, i^*) = p_{\pi^*} \cdot r^*[t + \mu_{\pi^*}] + q_{\pi^*} \cdot -\infty = -\infty$$.

- Suppose that $$i^* = N + 1$$, and $$t \leq t^{\text{stop}}$$; then:
– If \( \pi = () \), then from the definitions of \( g \) and \( g^* \) it is clear that:
\[
g(t, \pi^*) = g^*(t, i^*) = 0.
\]

– Otherwise:
\[
\begin{align*}
\text{\( \pi \notin A \) and } r^n[t + \mu_x] &\leq 0 \text{ then, } \forall \pi, \ g(t + \mu_{\pi(1)}, \pi(2 :)) \leq 0. \text{ This is due to } \\
\text{the fact that } c_1 \text{ is a non-decreasing function. Clearly in this case } \pi^* = (), \text{ this is a contradiction.} \\
\end{align*}
\]

• Suppose that \( i < N + 1 \) then:

– If \( \pi = () \), then it is clear from the definition that \( g(t, \pi^*) < 0 \). This implies that
\[
\text{\( \forall x \in A, \ r^n[t + \mu_x] > 0 \), where such an } x \text{ to exist } g(t, \pi^*) < g(t, (x)) \text{ which is a contradiction.}
\]

– Otherwise:
\[
\begin{align*}
\text{\( \pi \notin A \) and } r^n[t + \mu_x] &\leq 0 \text{ then, } \forall \pi, \ g(t + \mu_{\pi(1)}, \pi(2 :)) \leq 0. \text{ This is due to } \\
\text{the fact that } c_1 \text{ is a non-decreasing function. Clearly in this case } \pi^* = (), \text{ this is a contradiction.} \\
\end{align*}
\]

• Suppose that \( \pi = () \), then from the definitions of \( g \) and \( g^* \) it is clear that:
\[
g(t, \pi^*) = g^*(t, i^*) = 0.
\]

Using our inductive assumption we can see that \( g(t' + 1, \pi^*) = g^*(t' + 1, i) \) iff \( \pi = \pi(1) \), but this is true from assumption 2-\( \mathcal{A} \).
Suppose that our claim is true for \( i \geq k \). If \( i = k - 1 \) then:

\[
g^*(t, k - 1) = p_{k-1} \cdot r^* [t + \mu_{k-1}] + q_{k-1} \cdot \max_{j \in \{k, \ldots, N+1\}} \left( g^*(t + \mu_{k-1}, j) \right)
\]

\[
g(t, \pi) = \prod_{\pi \in \Pi(1)} \cdot r^* [t + \mu_{\pi(1)}] + q_{\pi(1)} \cdot g(t + \mu_{\pi(1)}, \pi(2 :))
\]

By our inductive assumption, and by assumption 2-\( \mathcal{A} \) we see that

\[
g(t, \pi) = g^*(t, \pi)
\]

Lemmas 7.2, 7.3, 7.4, 7.5, and theorem 7.1 show that the solution maximizing \( \hat{f} \) can be recovered once \( g^*(t, x) \) is known.
Chapter 8

Calculating the Upper Bound

8.1 Calculating the Upper Bound

In our computation we define a two dimensional array of variables $Y(x,i)$. In proof 8.1 we show that after executing algorithm 6, variable $Y(x,i) = g^*(x,i)$. Then from section 7.1.7 we have that
\[
\hat{f} \left( \text{Opt} \left( \hat{P}(A, t^{max}) \right) \right) = \max_{i \in A} g^* (t_{stop}, i),
\]
and that an upper bound for our original problem is $\max_{i \in \{1, ..., N\}} Y(t_{stop}, i)$. Lastly, algorithm 6 shows how the solution can be recovered.

**Theorem 8.1.** Algorithm 6 correctly calculates $g^*(t_{stop}, N)$, and terminates after executing a finite number of calculations.

**Proof.**

- Case 1: When $x < \mu_i$, the algorithm calculates $g^*(x, i)$ explicitly.
Algorithm 6 UpperBound

1: procedure UpperBound($\Omega, t^{stop}$)
2:   $R(1, ..., |\Omega|) \leftarrow \text{SortByRevenue}(\Omega)$  \hspace{1cm} $\triangleright j > i \implies r_{R(i)} \leq r_{R(j)}$
3:   $Y(t^{stop}, |\Omega|)$  \hspace{1cm} $\triangleright$ Declare a 2-dimensional variable array.
4:   for $x \in (0, ..., t^{stop})$ do
5:     for $i \in (1, ..., |\Omega|)$ do
6:       $X^d \leftarrow \min(\mu^{f}_{R(i)}, \mu^{s}_{R(i)})$  \hspace{1cm} $\triangleright$ Declare and initialize the variable.
7:       if $x < X^d$ then
8:         $Y(x, i) \leftarrow -\infty$
9:       else if $x = X^d$ then
10:          $Y(x, i) \leftarrow p_{R(i)} \cdot r_{R(i)}$
11:       else
12:          $X^v \leftarrow 0$  \hspace{1cm} $\triangleright$ Declare and initialize the variable.
13:          for $j \in (i + 1, ..., |\Omega|)$ do
14:            if $x > X^d$ $\land$ $X^v < Y(x - X^d, j)$ then
15:              $X^v \leftarrow Y(x - X^d, j)$
16:            end if
17:          end for
18:          $Y(x, i) \leftarrow p_{i} \cdot r_{R(i)} + (1 - p_{R(i)}) \cdot X^v$
19:       end if
20:     end for
21:   end for
22: return $\text{Trace}(\Omega, t^{stop}, R, Y)$
23: end procedure

- Case 2: When $x = \mu_i$, the algorithm calculates $g^*(x, i)$ explicitly.

- Case 3: When $x > \mu_i$, each of $Y(x - \mu_i, j)$ $\forall x \in (0, ..., t^{stop}), \forall j \in (i + 1, ..., |\Omega|)$ will have been calculated already. The fact that $\mu_i \in \mathbb{R}_{>0}$, and the order of the loops ensure that these values will have been calculated at the time the calculation for $Y(x, i)$ is made.

Thus, algorithm 6 correctly calculates $g^*(t^{stop}, N)$. Algorithm 6 terminates after executing a finite number of calculations because $t^{stop}$ and $|\Omega|$ are both finite. \qed
Algorithm 7 Trace

1: procedure Trace(\mathcal{Q}, t, R, Y)  \triangleright O(|\mathcal{Q}| \cdot \log(|\mathcal{Q}|)) complexity.
2:     \mathcal{X} \leftarrow \{\}
3:     X^v \leftarrow 0
4:     X^d \leftarrow 0
5:     X^a \leftarrow \emptyset
6:     z \leftarrow t
7:     do
8:         for i \in (1, \ldots, |\mathcal{Q}|) do
9:             if X^v < Y(z, i) then
10:                 X^v \leftarrow Y(z, i)
11:                 X^d \leftarrow \min(\mu^f_{R(i)}, \mu^s_{R(i)})
12:                 X^a \leftarrow R(i)
13:             end if
14:         end for
15:         if X^v > 0 \land X^a \notin \mathcal{X} then
16:             \mathcal{X} \leftarrow \mathcal{X} \cup \{X^a\}
17:             z \leftarrow z - X^d
18:         end if
19:     while X^v > 0
20:     return (X^v, \text{SortByRevenue}(\mathcal{X}))
21: end procedure
Chapter 9

Indexing and Compression

9.1 Indexing and Compression

Finding a nearest neighbor state (relative to some original state) quickly is non-trivial because the problem involves a search along two dimensions time $t$ and remaining feasible “potentially profitable” actions $\mathcal{Q}$. The importance of nearness along the dimensions are not equal, a solution that is “near” along the time dimension may be infeasible because the optimal action for the neighbor my not be feasible or not “potentially profitable” in the original state (it is not in set $\mathcal{Q}$). In contrast we expect that an optimal action from a state with an identical set of feasible, “potentially profitable” actions $\mathcal{Q}$ to the original state but where slightly more or less time has elapsed than in the original state will in-fact often be an optimal action for the original state.

This intuition implies that the set of feasible, “potentially profitable” actions $\mathcal{Q}'$ of a nearest neighbor of a state with feasible, “potentially profitable” actions $\mathcal{Q}$ is the maximum available subset of actions. That is $\mathcal{Q}' \subseteq \mathcal{Q}$, $|\mathcal{Q}'| \geq |\mathcal{X}|$ $\mathcal{X} \subseteq \mathcal{Q}$ $\forall \mathcal{X} \subseteq \hat{\mathcal{Q}}$ where $\hat{\mathcal{Q}}$ represents the set of all subsets of $\mathcal{Q}$ that have been pre-calculated in our approximation algorithm 4. Once
a nearest neighbor state with a feasible, “potentially profitable” set of actions has been identified the optimal action from that state will have been pre-calculated for all time values \( t \in (0, ..., t_{\text{stop}}) \). In other words, while our “nearest neighbor” query along the actions domain is approximate, our query along the time dimension will be exact.

Let \( \hat{S} \) represent the set of states which have been evaluated by \( RTDP \). Then let \((\hat{w}, \hat{t}, \hat{Q}) = \hat{s} \in \hat{S} \) where \( \hat{t} \) represents the set of \( t \) values evaluated by \( RTDP \), and \( \hat{Q} \) represents the sets of feasible, “potentially profitable” actions sets evaluated by \( RTDP \).

We address the problems of finding nearest neighbor states along each of these dimensions separately. Given a state of a state \( s = (w, t, Q) \) we first find a “nearest neighbor” feasible, “potentially profitable” action set \( Q' \in \hat{Q} \). Then we locate and return \( s' = (w', t, Q') \in \hat{S} \).

In our solutions we make use of the well known binary search tree data structure [24]. We use without definition the operations \( \text{Insert}(\tau, X), \text{Find}(\tau, X), \text{FindFloor}(\tau, X) \), and \( \text{FindCeil}(\tau, X) \) where \( \tau \) represents a binary tree, and \( X \) represents a data item such that a total ordering among all such data items exists. The operation \( \text{FindFloor}(\tau, X) \) returns the value \( X \) if such a value has previously been inserted into the tree, otherwise it returns the in-order predecessor of \( X \) in \( \tau \). Similarly, \( \text{FindCeil}(\tau, X) \) returns the value \( X \) is such a value has previously been inserted into the tree, otherwise it returns the in-order successor of \( X \) in \( \tau \). Each of these procedures is known to take time logarithmic in the height of the tree. In our cases the tree will be complete and thus the height of each tree will be \( \log_2(X) \) where \( X \) represents the number of nodes in the tree.

### 9.1.1 Action Dimension

We wish to find a “nearest neighbor” set of remaining feasible, “potentially profitable” actions \( Q' \) for a query set of remaining feasible, “potentially profitable” actions \( Q \). We define nearest
neighbor in this context to mean “the largest subset $Q'$ of $Q$” such that $Q' \in \hat{Q}$. Do do this we will build a binary tree index. A node $\eta_1$ in our index has is a tuple $(\kappa_1, \xi_1)$, where $\kappa_1 \in \{1, ..., N\}$ and $\xi \subseteq \hat{Q}$.

Intuitively, a “nearest neighbor” query with argument $Q$ into our tree index represents a path of no more than $N$ transitions. Each transition represents descending from a node $\eta_i$ in our index to one of it’s child nodes. The left child of node $\eta_i$, let’s call it $\eta_j$ has the property that $\kappa_i \notin x \forall x \in \xi_j$. The right child of node $\eta_i$, let’s call it $\eta_k$ has the property that $\kappa_i \in x \forall x \in \xi_k$.

No path initiating from the root of our tree will contain duplicate $\kappa$ values, thus the height of the tree is no larger than $N$. For a given query if the terminal node in the path representing the query is $\eta_i$ then $\xi_i$ represents the set of “nearest neighbors” states, and we may choose a state from this set arbitrarily.

We give the details for constructing a complete tree with these properties in algorithms 8, 9, and 10.

Binary trees are defined on elements with a strict ordering, in our case this ordering is not obvious and so we state it explicitly: $\eta_i < \eta_j \iff \kappa_j \notin x \mid \exists x \in \xi_i$.

Finally, notice that when the disjoint sets in our algorithms are represented with the union-find data-structure [39] we perform both a union of two sets, and the selection of sets containing a certain element in essentially constant time.

### 9.1.2 Time Dimension

During algorithm 4 we make use of algorithm 6 to gradually tighten the optimal path from our initial to some final states. Algorithm 6 in turn relies on algorithm 7 to retrieve the
Algorithm 8 PickPivot

1: procedure PickPivot(\(\mathcal{Z}, M\))

2: \(Y()\) \hfill \triangleright \text{Declare } Y \text{ as a 1-dimensional dynamically re-sizable variable array.}

3: \(\text{for } i \in \mathcal{Z} \text{ do}\)
4: \(X^l \leftarrow 0\)
5: \(X^r \leftarrow 0\)
6: \(X^v \leftarrow \infty\)
7: \(X^a \leftarrow \emptyset\)
8: \(\text{for } j \in M \text{ do}\)
9: \(\text{if } i \subseteq j \text{ then}\)
10: \(X^l \leftarrow X^l + 1\)
11: \(\text{else}\)
12: \(X^r \leftarrow X^r + 1\)
13: \(\text{end if}\)
14: \(\text{end for}\)
15: \(\text{if } |X^r - X^l| < X^v \text{ then}\)
16: \(X^v = |X^r - X^l|\)
17: \(X^a = i\)
18: \(\text{end if}\)
19: \(\text{end for}\)
20: \(\text{return } X^a\)
21: end procedure

optimal value and decision from sub-problems solved in algorithm 4 (optimal with regard to the originating state passed as a parameter to algorithm 6). In fact the sub-problem solved by algorithm 4 are sufficient to give the optimal value and decision for any state with the same feasible, “potentially profitable” task set \(\mathcal{Q}\) as passed to algorithm 4 and with any time value \(t' \in (0, \ldots, t)\) (where \(t\) is a component of the state passed to algorithm 4. Notice that while algorithm 4 has complexity \(O(t^{\text{stop}} \cdot N)\), algorithm 7 has complexity \(O(|\mathcal{Q}| \cdot \log(|\mathcal{Q}|))\).

In fact we can replace the following line from algorithm 4:

1: \(\text{return TRACE}(\mathcal{Q}, t, R, Y)\)
Algorithm 9 BuildTreeActions

1: procedure BuildTreeActions($\mathcal{Z}, M, \tau$) \hspace{1cm} $\triangleright$ $\tau$ represents the binary tree index.
2: $X \leftarrow \text{PickPivot}($\mathcal{Z}, M$)$
3: if $|\mathcal{Z}| = 0$ then
4: \hspace{0.5cm} return $\tau$
5: else if $|M| = 1$ then
6: \hspace{0.5cm} Insert$(\tau, m \in M)$
7: \hspace{0.5cm} return $\tau$
8: else
9: $X^l \leftarrow \{\}$ \hspace{1cm} $\triangleright$ $X^l$ is a set.
10: $X^r \leftarrow \{\}$ \hspace{1cm} $\triangleright$ $X^r$ is a set.
11: for $i \in M$ do
12: \hspace{0.5cm} if $X \subseteq i$ then
13: \hspace{1cm} $X^r \leftarrow X^r \cup i$
14: \hspace{0.5cm} else
15: \hspace{1cm} $X^l \leftarrow X^l \cup i$
16: \hspace{0.5cm} end if
17: \hspace{1cm} end if
18: end for
19: $\eta \leftarrow (X, M)$
20: Insert$(\tau, \eta)$
21: $X^l_k \leftarrow \bigcup_{x^l \in X^l} \{a \mid a \in x^l\}$ \hspace{1cm} $\triangleright$ $X^l_k$ is a set.
22: $X^r_k \leftarrow \bigcup_{x^r \in X^r} \{a \mid a \in x^r\}$ \hspace{1cm} $\triangleright$ $X^r_k$ is a set.
23: $X^l_k \leftarrow \mathcal{Z} \cap X^l_k$
24: $X^r_k \leftarrow (\mathcal{Z} \cap X^r_k) - \{X\}$
25: BuildTreeActions$(\tau, X^l_k, X^l)$
26: BuildTreeActions$(\tau, X^r_k, X^r)$
27: return $\tau$
28: end if
29: end procedure

With these lines:

1: for $i \in (0, ..., t)$ do
2: \hspace{0.5cm} $J^v(i), J^a(i) \leftarrow \text{Trace}(Q, t, R, Y)$
3: \hspace{0.5cm} return $J^v, J^a$
4: end for
Algorithm 10 BuildActionTreeIndex

1: procedure BuildActionTreeIndex(\(\hat{Q}\))
2: \(\tau \leftarrow \text{BinaryTree}(\)\)
3: return BuildTimeTree(\(\bigcup_{\exists \in \hat{Q}} \{a \mid a \in \Omega\}, \hat{Q}, \tau\))
4: end procedure

The returned vector variables encode the solutions for states with feasible, “potentially profitable” task sets \(\Omega\) and times \(t' \in (0, \ldots, t)\). Thus we may modify algorithm 4 as follows:

- Each time that a call to SimulateAttempt generates a state vector whose feasible, “potentially profitable” task sets \(\Omega\) has not yet been encountered by algorithm 4 we will execute a call to UpperBound(\(\emptyset, t^{stop}, \Omega\)) (where UpperBound has been modified as above) and store the returned solution vectors indexed with the key \(\Omega\). Finally, we will use the values at index \(t\) of these solution vectors to proceed with our simulations as before.

- Each time that a call to SimulateAttempt generates a state vector whose feasible, “potentially profitable” task sets \(\Omega\) has been encountered we fetch the previously calculated values from our index and return the solution for the appropriate value of \(t\).

The end result of these modifications is that our UpperBound problem is solved at most \(|\hat{Q}|\) times, at the expense of maintaining an index keyed to \(\hat{Q}\) and \(O(t^{stop} \cdot |\hat{Q}|)\) additional space. Our modifications do not modify the time complexity of algorithm 4.

Next we address the problem of constructing an optimally balanced binary tree index keyed to \(\hat{Q}\). After algorithm 4 terminates we will construct indices on both the time domain \(t \in (0, \ldots, t^{stop})\) and the domain of feasible, “potentially profitable” tasks. To construct the index on the time domain of each feasible, “potentially profitable” set of tasks \(\Omega \in \hat{Q}\) we
will traverse the arrays of optimal actions and action values returned for that set by our modification of algorithm 7, given a particular \( Q \) we will refer to these variable arrays as \( J^u(i), J^a(i) \) where \( i \in (0, ..., t^{stop}) \) as in our modification above.

We build a complete binary tree keyed on \( i \in (0, ..., t^{stop}) \) with values \( J^a(i) \). Our tree index will compress sequences contiguous (in the key) of identically valued terms. For example if

\[
J^a(1) = 5, J^a(2) = 5, J^a(3) = 7, J^a(4) = 7, J^a(5) = 9
\]

our binary tree index will only need to store values for

\[
J^a(1) = 5, J^a(3) = 7, J^a(5) = 9.
\]

Algorithm 13 gives the procedure for building our time index for any variable array \( J^a \), recall that after our modification algorithm 4 generates one such array for each encountered feasible, “potentially profitable” task set \( Q \in \hat{Q} \). Algorithm 11 compresses contiguous action sequences, and algorithm 12 builds the tree index after the condensing.

We illustrate the tree construction process graphically in figure 9.1. In our figure each color represents a different task.

![Binary Time Tree](image.png)

**Figure 9.1: Binary Time Tree**
Algorithm 11 IntervalReduce

1: procedure INTERVALREDUCE($J^a$)
2:     $Y()$ \Comment{Declare $Y$ as a 1-dimensional dynamically re-sizable variable array.}
3:     $X \leftarrow \emptyset$
4:     for $i \in (0, \ldots, t_{stop})$ do
5:         if $J^a(i) \neq X$ then
6:             $Y(i) \leftarrow J^a(i)$
7:             $X \leftarrow J^a(i)$
8:         end if
9:     end for
10:    return $Y$
11: end procedure

9.1.3 Index Use and Complications

Notice that for the static problem building the indices as we’ve described serves no purpose. All states encountered by algorithm 4 when the problem being solved is the static ordering problem are evaluated within the algorithm. Section 5.2 describes a two step process wherein solutions are pre-calculated and then looked-up during the “in-use” phase. In section 5.2 optimal actions were looked-up in a pre-calculated table during the “in-use” phase, the complexity of these queries is $O(1)$. In this section we have shown how approximately optimal actions can be indexed in complete binary trees, the time complexity of the query in the index along the action dimension is $O(N)$, and along the time dimension is $O(\log(t_{stop}))$. While the cost of the indexing query along the time dimension is reasonable the cost along the action dimension may be prohibitive for large values of $N$, and very demanding time constraints for the “in-use” phase.

A further complication is that the nearest neighbor set of actions $Q'$ returned by a query into the action index may be a neighbor that is too unlike the query state to give useful results. This observation leads to a family of strategies parameterized by parameter $\lambda$ where $\lambda$ indicates the minimum distance between two feasible, “potentially profitable” action sets.
Algorithm 12 BuildTimeTree

1: procedure BuildTimeTree(Y, τ)  ▷ τ represents the binary tree index.
2:   X ← ⌊|Y|/2⌋
3:   if |Y| = 0 then  ▷ Let |Y| indicate the number of elements in the array.
4:     return τ
5:   else if |Y| = 1 then
6:     Insert(τ, Y(1))
7:   else
8:     Insert(τ, Y(X))
9:     Insert(τ, Y(1, ..., X))  ▷ Let Y(1, ..., X) indicate a sub-sequence of 1-dimensional array Y.
10:    Insert(τ, Y(X + 1, ..., |Y|))
11: end if
12: return τ
13: end procedure

Algorithm 13 BuildTimeTreeIndex

1: procedure BuildTimeTreeIndex(Ja)
2:   Y() ← IntervalReduce(Ja)
3:   τ ← BinaryTree()
4:   return BuildTimeTree(Y, τ)
5: end procedure

for which the suggested action is attempted. When the distance between the query feasible, “potentially profitable” action set and it’s nearest neighbor is above λ no action is taken and execution terminates (this is to eliminate the threat of penalties resulting from excessive delay). A distance between feasible, “potentially profitable” action sets Q and Q’ can simply be \( \text{dist}(Q, Q') = ||Q| - |Q'|| \). The strategy corresponding to \( \lambda = 0 \) states that if an exact match nearest neighbor set of actions is not found at query time then no action. This \( \lambda = 0 \) strategy is also the strategy we propose for cases where \( N \) too large and time constraints too demanding to allow the use of the action index during the “in-use” phase.
Chapter 10

Delay Distributions

10.1 Selecting a Delay Distribution

Our formulation up to this point has made no mention of the distributions used to model timing delays. Indeed an operator may use distributions of their choosing and apply our work with these choices. However, for the purposes of evaluating our work empirically we made a considered choice in picking a distribution for task delay times. Our choice of distribution stems from the work of [1, 15, 21, 23, 41] in identifying the distribution as one that closely models response-times in many varied settings. As noted in [15] and [21] a Weibull distribution often measures delay times even more accurately than a lognormal distribution and is another typical (perhaps even more typical choice) for approximating delays. However as noted in [21] lognormal a distribution can be used to approximate a variable more accurately represented by a Weibull distribution with limited loss of accuracy. We choose to use the lognormal distribution rather than the Weibull distribution because the convolution of lognormal distributions can be more easily approximated than the convolution of Weibull distributions.
We assume that the delay times for both successful and unsuccessful task attempts follow log-normal distributions (though the distributions for success and failure of each product are distinct). This distribution is chosen as it models several aspects of internet delay times, in particular it ensures that negative delays have 0 probability, and that excessively large delays are always possible though not probable (it has the long tail property).

A random variable $X$ is lognormally distributed if $\log(X)$ is normally distributed. Lognormal variables are parametrized by a location term (usually referred to as $\mu$) and a scale term (usually referred to as $\sigma$) However these parameters can be obtained from the arithmetic mean and the arithmetic variance if those are known. In the example below and elsewhere in this paper (when in reference to a lognormal distribution) we use $\mu$ and $\sigma$ to refer to the arithmetic mean and arithmetic standard deviation not the location and shape of the distribution. So for example $\mu_a$ and $\sigma_a$ refer to the arithmetic mean and standard deviation of the lognormal distribution representing the delay for attempting task $a$.

$Pr(T_s^i + \sum_{j<i} T_f^j \leq t_{max})$ is evaluated as the cumulative distribution function of the lognormal distribution.

Where $Pr(T_s^i + \sum_{j<i} T_f^j \leq t_{max})$ is evaluated as the cumulative distribution function of the log-normal distribution with mean $\mu_i^s + \sum_{j<i} \mu_j^f$ and standard deviation $\sqrt{\sigma_i^s + \sum_{j<i} \sigma_j^f}$. A log-normal distribution with this derived mean and standard deviation approximates the convolution of the distributions $T_s^i$ and $T_f^j$ for all $j < i$. We treat $Pr(T_s^i + \sum_{j<i} T_f^j > t_{max})$ similarly.
Chapter 11

Greedy is not Enough

11.1 A Greedy Heuristic and How Greedy Can Fail

One criticism to our approach is that a simpler method could produce results as good or nearly as good as our tabulated calculation of expected values for the system states as we have defined them. While it’s difficult to combat such a broad criticism directly, in this section we show that a myopic policy can do badly relative our proposed policy.

In this example we simplify our trivial myopic policy would be to take no account of delay times at all, and from a state \( s \) to choose the highest expected value task \( a \) as calculated by \( \mathbb{E}[V(s,a)] = r_a \cdot p_a \). It should be clear that this policy can easily be made to perform arbitrarily badly. For example if the task chosen according to this policy from state \( s \) was task \( d \) and we know that \( \mu_d^* = M \cdot t^{\text{max}} \) then by increasing \( M \) we can increase the cost for choosing state \( d \) to an arbitrarily large value.

A simple myopic policy that still takes most relevant and available variables into account is one which will choose the next task at time \( t \) by ranking tasks by the expected value of their
success only and choose the task with the highest such rank. To simplify the example we assume that $t^{max}$ must not be exceeded.

The proposed myopic policy will suppose that the value of executing action from state $s$ is:

$$V(s) = p_d \cdot r_d \cdot Pr(t^{max} - t \leq t_{a}^{stop})$$

(11.1)

Where $s = (w, t, Q)$, and $d = \max_q(r_q \cdot p_q \cdot Pr(t^{max} - t \leq t_{q}^{stop}))$.

To calculate the term $Pr(t^{max} - t \leq t_{a}^{stop})$ we must choose a delay distribution. We choose to model the delays of both success and failure in this example (and in our empirical evaluation) as log-normal random variables.

Suppose $\mathcal{A} = \{1, 2, 3\}$, $t^{max} = 20$, $p_1 = p_2 = p_3 = 0.33$, $r_1 = 1.01, r_2 = 0.99, r_3 = 1.00$, $\mu_1^f = 1.95, \sigma_1^f = 0.05, \mu_1^s = 1.00, \sigma_1^s = 0.05, \mu_2^f = 0.01, \sigma_2^f = 0.05, \mu_2^s = 0.10, \sigma_2^s = 0.05, \mu_3^f = 1.00, \sigma_3^f = 0.15, \mu_3^s = 1.00, \sigma_3^s = 0.05$.

Reasoning about this example we can see that if we do not take the delay times into account then the expected values of the tasks ($r_a \cdot p_a$) would be very close. In fact if we were to disregard delay we would rank the tasks in order of decreasing value as 1, 3, 2. As in the trivial example above we can see that task 1 would be a poor choice regardless of the amount of time available as given it’s high arithmetic mean and low variance we are likely to expend all our available time while attempting task 1 while achieving a nearly identical expected revenue (when not accounting for delay) as if we attempted either task 2 or 3. By choosing task 1 should our attempt not succeed we will also be very unlikely to have enough time to attempt any remaining task. Intuitively it seems that a reasonable solution for this problem
will attempt task 3 when a large amount of time is available and then attempt task 2 should
the attempt at task 3 fail within a reasonably short period of time.

In table 11.1 we calculate the value of several states according our proposed valuation as
described in equation 4.1 (labelled $V_{4,1}$) and the value as calculated using the valuation of
the simple heuristic described in equation 11.1 (labelled $V_{11,1}$). We also give the task that
would be chosen from each of several when either of the proposed policies is used (labelled
$D_{4,1}$ and $D_{11,1}$ respectively).

<table>
<thead>
<tr>
<th>State</th>
<th>$V_{4,1}$</th>
<th>$D_{4,1}$</th>
<th>$V_{11,1}$</th>
<th>$D_{11,1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\emptyset, 1, {1, 2, 3})$</td>
<td>0.1938</td>
<td>2</td>
<td>0.1938</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 2, {1, 2, 3})$</td>
<td>0.3122</td>
<td>2</td>
<td>0.3122</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 3, {1, 2, 3})$</td>
<td>0.3250</td>
<td>2</td>
<td>0.3245</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 4, {1, 2, 3})$</td>
<td>0.3265</td>
<td>2</td>
<td>0.3265</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 5, {1, 2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 6, {1, 2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 7, {1, 2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 8, {1, 2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 9, {1, 2, 3})$</td>
<td>0.3308</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 10, {1, 2, 3})$</td>
<td>0.4389</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 11, {1, 2, 3})$</td>
<td>0.5420</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 12, {1, 2, 3})$</td>
<td>0.5491</td>
<td>2</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 13, {1, 2, 3})$</td>
<td>0.5495</td>
<td>2</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 14, {1, 2, 3})$</td>
<td>0.5497</td>
<td>2</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 15, {1, 2, 3})$</td>
<td>0.5498</td>
<td>2</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 16, {1, 2, 3})$</td>
<td>0.5499</td>
<td>2</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 17, {1, 2, 3})$</td>
<td>0.5504</td>
<td>3</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 18, {1, 2, 3})$</td>
<td>0.5627</td>
<td>3</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Pr</td>
<td>m</td>
<td>k</td>
<td>L</td>
</tr>
<tr>
<td>----------------</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>$(\emptyset, 19, {1, 2, 3})$</td>
<td>0.5998</td>
<td>3</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 20, {1, 2, 3})$</td>
<td>0.6492</td>
<td>3</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>$(\emptyset, 1, {2, 3})$</td>
<td>0.1938</td>
<td>2</td>
<td>0.1938</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 2, {2, 3})$</td>
<td>0.3122</td>
<td>2</td>
<td>0.3122</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 3, {2, 3})$</td>
<td>0.3250</td>
<td>2</td>
<td>0.3250</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 4, {2, 3})$</td>
<td>0.3265</td>
<td>2</td>
<td>0.3265</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 5, {2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 6, {2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 7, {2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 8, {2, 3})$</td>
<td>0.3267</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 9, {2, 3})$</td>
<td>0.3308</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 10, {2, 3})$</td>
<td>0.4378</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 11, {2, 3})$</td>
<td>0.5398</td>
<td>2</td>
<td>0.3267</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 12, {2, 3})$</td>
<td>0.5469</td>
<td>2</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 13, {2, 3})$</td>
<td>0.5473</td>
<td>2</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 14, {2, 3})$</td>
<td>0.5481</td>
<td>3</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 15, {2, 3})$</td>
<td>0.5488</td>
<td>3</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 16, {2, 3})$</td>
<td>0.5489</td>
<td>3</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 17, {2, 3})$</td>
<td>0.5489</td>
<td>3</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 18, {2, 3})$</td>
<td>0.5489</td>
<td>3</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 19, {2, 3})$</td>
<td>0.5489</td>
<td>3</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 20, {2, 3})$</td>
<td>0.5489</td>
<td>3</td>
<td>0.3300</td>
<td>3</td>
</tr>
<tr>
<td>$(\emptyset, 1, {1, 2})$</td>
<td>0.1938</td>
<td>2</td>
<td>0.1938</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 2, {1, 2})$</td>
<td>0.3122</td>
<td>2</td>
<td>0.3122</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 3, {1, 2})$</td>
<td>0.3250</td>
<td>2</td>
<td>0.3250</td>
<td>2</td>
</tr>
<tr>
<td>$(\emptyset, 4, {1, 2})$</td>
<td>0.3265</td>
<td>2</td>
<td>0.3265</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 11.1: Heuristic Example Results

Table 11.1 shows that following the policy advocated in equation 4.1 from initial state $(\emptyset, 20, \{1,2,3\})$ we are most likely to attempt tasks in the order $\{3,2,1\}$ with expected value $0.6492$. Following the myopic policy from equation 11.1 we are most likely to attempt tasks in the order $\{1,3,2\}$ with expected value (as calculated by the myopic approximation) $0.3333$. When calculated exactly (with equation 4.1) the value of attempting tasks in the sequence $\{1,3,2\}$ is $0.3501$. Thus in this example the percent difference of the expected
value of following these policies is 59.86%, following the policy advocated in equation 4.1 will produce an expected value of nearly twice that of following the simple myopic policy!
Chapter 12

Experimental Results

12.1 Empirical Evaluation

In this section we present empirical results from tests run using the non-convex penalty function from equation 2.3. Using real data obtained from a successful ad network we evaluated the performance of our policy over many simulated runs over each of a number of test instances. Our test suit is composed of several instances, each of which describes the set of applicable advertisers for a given publisher. Each instance is best interpreted from the point of view of a publisher. From this perspective each instance contains data relevant to choosing the next advertiser to solicit given the amount of elapsed time since the content has begun to load.

Each instance in our suite is identified by the tuple \((\text{publisher\_id}, t^{max})\), \(\text{publisher\_id}\) uniquely identifies the publisher and equivalently the instance. As described in the previous sections \(t^{max}\) is the maximal allowable servicing-time specified between the publisher and their ad network. Each advertiser \(a\) described in each instance is identified by a tu-
ple \((\text{advertiser\_id}, r_a, p_a, \mu_f^a, \mu_s^a, \sigma_f^a, \sigma_s^a)\). \text{advertiser\_id} uniquely identifies the advertiser, \(r_a, p_a, \mu_f^a, \mu_s^a, \sigma_f^a, \sigma_s^a\) are described in section 2.1.

We compare the performance of \text{GRASP\_OBOS} relative to a suite of heuristics used in practice and for small instances, and relative to the optimal dynamic programming solutions. We evaluate the efficacy of each policy by taking the average net revenue resulting from executing the policy over 10000 realized paths. A realized path refers to the state transitions, and the collection of realized samples drawn from random variable distributions during the process of a simulated ad request. Following a realized path in our simulation entails sampling from distributions representing the success or failure of each solicitation of a given advertiser, and the success or failure time of that solicitation. A realized path terminates either when we successfully solicit an advertiser (and collect the associated revenue), or when we exhaust our time budget for the publisher \((t > t_{max})\).

The success or failure of any given solicitation for a particular publisher is drawn from a Bernoulli distribution specific to that advertiser \(a\), and characterised by the sufficient statistic \(p_a\). The delay time of failure or success for a particular advertiser is drawn from a lognormal distribution characterized by the sufficient statistics \((\mu_f^a, \sigma_f^a)\) and \((\mu_s^a, \sigma_s^a)\) respectively. Lognormal distributions are commonly used to model failure and delay times of both natural, and human processes, see for example [1, 15, 21, 41, 23, 38].

Two common solutions to this problem are to order advertisers in order of non-increasing revenue (this is the optimal order if the content load time was not a consideration) and to order them at random. One reason that these approaches are commonly used is that they not require collecting and aggregating the data necessary to characterize the probabilities of success, or the delay times. In our figures these approaches are labeled ‘GMR’(greedy, maximal revenue) and ‘RANDOM’ respectively.
If the data necessary to make the characterizations were collected, an initially appealing approach would be to choose the next advertiser greedily, choosing the advertiser that maximizes the expected revenue of successfully soliciting a given advertiser (unsuccessful solicitations, and their delays are not considered); we label this strategy ‘GMER’ (greedy maximal expected revenue). Explicitly, if the time at evaluation is \( t \), for advertiser \( a \) the expected value is:

\[
p_a \left( r_a \cdot Pr(t + T^s_a \leq t_{max}) - c \left( t + T^s_a - t_{max}, a \right) \cdot Pr(t + T^s_a > t_{max}) \right)
\]

(12.1)

Notice that we can evaluate the probability terms in equation 12.1 from our Bernoulli success distributions and lognormal delay distributions (in the later case through the use of the cumulative density (CDF) function).

Finally, a slightly more sophisticated heuristic would be to greedily choose the next advertiser based on the weighted sum of two terms. One term representing the revenue of each advertiser \( r_a \), and the other representing the expected revenue of that advertiser (see equation 12.1). Let the weight coefficients be \( \alpha_1 = \frac{t}{t_{max}} \) and \( \alpha_2 = 1.0 - \alpha_1 \). Our heuristic valuation will choose the advertiser \( a \) maximizing the expression:

\[
\alpha_2 \cdot r_a + \alpha_1 \cdot p_a \left( r_a \cdot Pr(t + T^s_a \leq t_{max}) - c \left( t + T^s_a - t_{max}, a \right) \cdot Pr(t + T^s_a > t_{max}) \right)
\]

(12.2)

Equation 12.2 is a convex combination of our two terms. The intuition behind this heuristic is that we wish to choose the largest revenue terms early in our sequence (as would be optimal if delay time were not a factor), then as we approach the deadline we wish to take the
probability of succeeding within the remaining available time more and more into account. We refer to the policy represented by this heuristic (equation 12.2) as ‘GCC’ (greedy convex combination) in figures 12.1 and 12.2.

Each test instance in test suite 1 (figure 12.1) contains 10 applicable advertisers, this is a sufficiently small number that we may calculate the true optimal value for the problem by a dynamic programming formulation in a reasonable amount of time. Test suite 2 contains instances with advertisers set sizes of 11 and more (tests in the figure are non-decreasing in advertiser set size). Our exact dynamic programming solution method is similar to Richard Bellman’s approach to the traveling salesman problem in [9], we omit the details due to space limitations. The results given in figure 12.1 are average percent deviation from optimality for each test, the x-axis represents the value of the optimal solution. Table 12.1 aggregates these results across the tested instances.

In test suite 2 (figure 12.2) test instances represent sets of applicable advertisers of sizes between 11 and 15 (inclusive), the tests are non-decreasing in the number of applicable advertisers in each instance. An optimal value for tests in suite 2 could not be effectively computed by our dynamic program (this is an exponential time algorithm), thus rather than presenting the data as deviations from optimality we instead present the raw averages for each algorithm. Table 12.2 aggregates these results across the tested instances. Instances ‘test-110’ to ‘test-76’ have advertiser set size 11, ‘test-253’ to ‘test-92’ have advertiser set size 12, ‘test-16’ has advertiser set size 13, ‘test-129’ to ‘test-17’ have advertiser set size 14, and instances ‘test-239’ and ‘test-354’ has advertiser set size 15.

While our proofs of convergence and the $\epsilon$ optimality gap guaranteed by our implementation of the RTDP algorithm (labeled XRTDP in the figure below) are contingent on a convex penalty function we found that our algorithm performs quite well even with a non-convex penalty function commonly used in practice. In figure 12.3 we show that our RTDP imple-
Heuristic Average Percent Deviation

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Average Percent Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMR</td>
<td>-27.14</td>
</tr>
<tr>
<td>RANDOM</td>
<td>-36.03</td>
</tr>
<tr>
<td>GMER</td>
<td>-7.71</td>
</tr>
<tr>
<td>GCC</td>
<td>-7.62</td>
</tr>
<tr>
<td>GRASP_OBOS</td>
<td>-0.67</td>
</tr>
</tbody>
</table>

Table 12.1: Average Percent Deviation Over all Realized Paths, Over all Instances

Heuristic Average Net Revenue (CPM)

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Average Net Revenue (CPM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMR</td>
<td>0.149</td>
</tr>
<tr>
<td>RANDOM</td>
<td>0.146</td>
</tr>
<tr>
<td>GCC</td>
<td>0.255</td>
</tr>
<tr>
<td>GMER</td>
<td>0.249</td>
</tr>
<tr>
<td>GRASP_OBOS</td>
<td>0.268</td>
</tr>
</tbody>
</table>

Table 12.2: Average Net Revenue Over all Realized Paths, Over all Instances

Our implementation outperforms our GRASP algorithm (and is essentially optimal) for small instance over which we may calculate the value of the optimal solution.

We found in our experiments that the optimal value achievable from our initial state configuration converged to near optimality. Figures figs. 12.4 to 12.8 illustrate this convergence as our XRTDP algorithm progresses (the x-axis in these figures represents an iteration counter) for several specific test cases. In these figures the red line represents our upper bound solutions, the yellow line represents the true optimal value of the initial state, and the blue line represents the lower bound value of this initial state at a given iteration.
Figure 12.1: Advertisers sets of size 10
Figure 12.2: Advertisers sets of sizes 11, 12, 13, 14, 15
Figure 12.3: Performance of XRTDP

Figure 12.4: Convergence of XRTDP
Figure 12.5: Convergence of XRTDP

Figure 12.6: Convergence of XRTDP
Figure 12.7: Convergence of XRTDP

Figure 12.8: Convergence of XRTDP
Chapter 13

Other Applications

13.1 Other Applications

We believe that the problem and solution methods we have described are applicable to a broader context than online advertising. In this section we present an informal description of the problem, and give several possible alternative applications of our work.

13.2 An Informal Description

Suppose that a set of mutually exclusive tasks exist and that each task has an associated probability of success and associated reward. The question that arises in this situation is to determine the order of execution of the various tasks in order to receive the highest reward within a certain period of time. We present several examples of problems falling into this general class.
13.3 Exclusive Access Cheapest Paths

Suppose that \( k \) least cost paths (and their associated costs) between a particular origin and a particular destination in a connected communications network are calculated at the beginning of the day. Within the day at random times a dedicated communications circuit from the origin to the destination must be established, used, and finally terminated. The dedicated circuit will be used for an arbitrary amount of time (this can be called the lifetime of the circuit). We have a time budget \( t_{\text{max}} \) that limits the amount of time we have to set-up our circuit. In other words if a circuit must be established at a time no later than \( t + t_{\text{max}} \) where \( t \) represents the instant that we become aware of the intent to establish the circuit. In the graph representation of our network, routers and hosts are nodes, point to point communications links are edges, and dedicated communications circuits occur over paths between the origin and destination.

Each link in our communications network may be committed or in use at the time that we wish to establish a circuit from the origin to the destination (perhaps other users of the network are also establishing and using dedicated circuits at random times). To remove the uncertainty of whether a particular path is available at any given time we must first reserve each edge of our path before we can use the path as our circuit. When any edge of a path \( i \) from the origin to the destination is reserved or in use at the time the circuit is to be established path \( i \) is unavailable for the circuit and cannot be used. Suppose that at any time the status (reserved/in-use or available) of each edge in the graph representation of the communications network is known only to the nodes on which the edge is incident, and each edge may only be reserved for use at a node on which it is incident.

To establish a communications circuit using path \( i \) to be used at time \( t' < t + t_{\text{max}} \) it must be reserved for use at a time \( t'' \) where \( t < t'' < t' < t + t_{\text{max}} \). Each edge of path \( i \) must be reserved separately at one of it’s incident nodes. To be reserved an edge must not be
reserved or in use at the time when we wish to reserve it at its incident node, after we reserve it the edge cannot be reserved or used (other than in our circuit) while we have not released the edge. If all the edges of a path are not in use or reserved for a circuit other than our circuit then that path is available.

Ideally we would always choose a least cost path over which to set up our circuit. However, at the time the circuit is to be established any of the edges of the path over which it is to be established may be reserved or in use. Thus within the interval \((t, ..., t'')\) we must probe the network to determine and reserve the optimal available path (from our pre-calculated collection of \(k\) paths).

As edges can only be reserved at the nodes that they are incident on, to reserve a path we must send a computational agent to attempt to traverse it from origin to destination, in the process our agent will attempt to reserve each edge in the path for use by our circuit. If during this reservation process we encounter an edge that is not available for reservation the agent will return to the origin node along the same path, and release each edge that it was able to reserve. When a “dead-end” or “road block” like this is encountered we will have failed to reserve our intended path, and will try again to reserve a different path.

Paths in this model can be ranked by two metrics, path length (the number of edges composing the path), and path cost (the sum of edge costs). A cheaper path may be longer than a more expensive path (e.g. a path of 5 edges, each with cost 1, rather than a path composed of a single edge with cost 6). If edges were equally likely to be reserved or in use we would be more likely to be able to reserve a shorter path than a longer path since the probability of all edges in a path being available for reservation would decrease geometrically in the length of the path. If we suppose that the delay that our computational agent suffers when traversing any given edge is identical to the delay of traversing any other edge, then shorter paths will take longer to reserve (have higher expected response time) than longer paths.
Trying to reserve all $k$ shortest paths in parallel is unreasonable. Doing so would involve acquiring resources frivolously (e.g. we reserve 2 paths from origin to destination but set-up a circuit on only one of them). Furthermore, within the context of multiple user access to the communications network it is easy to see how multiple users acting to reserve paths all $k$ of their paths can result in deadlock, and significant delays for all users. In short, our solution to this problem must:

- Reserve an available path from our origin to our destination within the allowed time limit.
- Ensure that the chosen path is the cheapest such path that is available.
- In the process of reserving our path the algorithm should not frivolously reserve links that will not be used, or should release them as soon as the knowledge that the path is not available is known.
- Not depend on coordination with other users who may be reserving or using the links of the network.

To solve this problem we must sequentially attempt to reserve one of our pre-determined paths. Each time that a path cannot be fully reserved, we will release those edges of the path that we have been able to reserve, and attempt to reserve another path. Once a path is fully reserved it can be used for out communications circuit. We may be collecting statistics on our $k$ paths, such as how likely we are to be able to reserve each path, what is the average delay associated with reserving or failing to reserve each path. These statistics can help drive our decisions as to the order in which we will attempt to reserve the paths in.
13.4 Spot Market Contracting for Logistics Services

Shippers who need to procure logistics (for example truckload trucking) services typically have long term relationships with one or more core carriers. However, the prices negotiated with secondary or tertiary carriers might not be competitive. Therefore, if the shipper’s primary carrier for a lane (an origin-destination pair) does not have available capacity, that shipper will try to procure short term capacity on a spot market. One way to do this is to offer the load sequentially to a set of carriers and contract for the movement when an agreeable price/service match has been found. And, if the contract is not agreed upon quickly, the shipper could revert back to a secondary carrier with whom it has a long term contract.

13.5 Purchasing Real Estate

In a highly competitive real-estate markets such as the Silicon Valley in California or London, buyers submit sequential bids for properties and are typically interested in more than one property in the market. Homes regularly have many competing bids, which are evaluated in a sealed bid auction. Further, if buyers are relocating from a “colder market” to a “hotter one” they will often wait to look for a house in the hot market until they have a firm closing date in the colder one. Therefore they have tight time constraints for finding a new home or face the cost of relocating twice. If agents can estimate the likelihood than a specific bid will be successful, our method can assist them to order their bids.
Chapter 14

Conclusion

14.1 Conclusion

While our problem can be solved by dynamic programming and while the problem can be solved off-line, for sequences of more than 10 advertisers the DP is not tractable. Therefore we have developed a heuristic solution for the static ordering variation of this problem. The heuristic is an adaptation of the greedy randomized search procedure (GRASP) meta-heuristic([34]).

Our meta-heuristic solves the static ad sequencing problem efficiently and we have evaluated it on an extensive data set provided to us by an ad network. Empirical evaluation shows that GRASP OBOS is on average within 1% of optimal for problems which can be solved by dynamic programming. For larger problems we show that GRASP OBOS outperforms any of the heuristics we know of that are being used in practice.
Our RTDP based solutions achieve superior performance to our GRASP based solution, and give solutions that are optimal up to a user specified $\epsilon$ value when a convex penalty function is specified.
Bibliography


[27] H. B. McMahan, M. Likhachev, and G. J. Gordon. Bounded real-time dynamic pro-
malling: Rtdp with monotone upper bounds and performance guarantees. In Prog-


[29] V. Mirrokni and H. Nazerzadeh. Deals or no deals: Contract design for online advertis-
ing. Available at SSRN 2693037, 2015.


[33] W. B. Powell. Approximate Dynamic Programming: Solving the Curses of Dimension-


[35] B. Riordan-Butterworth. Openrtb api specifcation. Technical re-
port, Interactive Advertising Bureau, http://www.iab.net/media/file/

guaranteed delivery in online advertising. In Proceedings of the 21st ACM international

[37] T. Smith and R. Simmons. Focused real-time dynamic programming for mdps: Squeez-

[38] D. P. Strum, J. H. May, and L. G. Vargas. Modeling the uncertainty of surgical pro-
cedure times comparison of log-normal and normal models. The Journal of the American


[40] J. Turner. The planning of guaranteed targeted display advertising. Operations research,

[41] W. J. van der Linden. A lognormal model for response times on test items. Journal of


Appendix A

Proof of Theorem 2.1

A.1 Proof of Theorem 2.1

Proof. Jensen’s inequality states that for any convex function $\phi$ and any random variable $X$: $\phi(\mathbb{E}[X]) \leq \mathbb{E}[\phi(X)]$. We can rewrite this as:

$$-\mathbb{E}[\phi(x)] \leq -\phi(\mathbb{E}[X]). \tag{A.1}$$

As variables $X_i$ and $X_j$ are independent for any $i \neq j$: $\mathbb{E}[X_i \cdot X_j] = \mathbb{E}[X_i] \cdot \mathbb{E}[X_j]$. Also, $\mathbb{E}[\bar{X}_i \cdot \bar{X}_j] = \mathbb{E}[\bar{X}_i] \cdot \mathbb{E}[\bar{X}_j]$, and $\mathbb{E}[\bar{X}_i \cdot X_j] = \mathbb{E}[\bar{X}_i] \cdot \mathbb{E}[X_j]$. 

101
Applying Jenson’s inequality (with $\phi = c^1$):

$$
\mathbb{E}[\psi_{\pi(k)}] = \mathbb{E}[X_{\pi(k)}] \cdot \prod_{i=1}^{k-1} \left( \mathbb{E}[X_i] \right) \cdot \left( r_{\pi(k)} - \mathbb{E}\left[ c^1\left( t_{\text{max}}^{\pi(k)}, T_s^{\pi(k)} + \sum_{j=1}^{k-1} T_f^{\pi(j)} \right) \right] \right)
$$

$$
\leq \mathbb{E}[X_{\pi(k)}] \cdot \prod_{i=1}^{k-1} \left( \mathbb{E}[X_i] \right) \cdot \left( r_{\pi(k)} - c^1\left( t_{\text{max}}^{\pi(k)}, T_s^{\pi(k)} + \sum_{j=1}^{k-1} \mathbb{E}[T_f^{\pi(j)}] \right) \right)
$$

(A.2)

Applying the relabellings from equation 2.5 gives us:

$$
\mathbb{E}[\psi_{\pi(k)}] = p_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( 1 - p_{\pi(i)} \right) \cdot \left( r_{\pi(k)} - \mathbb{E}\left[ c^1\left( t_{\text{max}}^{\pi(k)}, T_s^{\pi(k)} + \sum_{j=1}^{k-1} T_f^{\pi(j)} \right) \right] \right)
$$

$$
\leq p_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( 1 - p_{\pi(i)} \right) \cdot \left( r_{\pi(k)} - c^1\left( t_{\text{max}}^{\pi(k)}, \mu_s(\pi(k)) + \sum_{j=1}^{k-1} \mu_f(\pi(j)) \right) \right)
$$

(A.3)

Applying the substitutions from section 2.5 into the optimization objective in equation 2.10 produces a problem the solution to which is an upper bound to our original problem:

$$
f(\pi) = \mathbb{E}[\rho(\pi)] = \mathbb{E}\left[ \sum_{k=1}^{N} \psi_{\pi(k)} \right]
$$

$$
= \sum_{k=1}^{N} \left( p_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( 1 - p_{\pi(i)} \right) \cdot \left( r_{\pi(k)} - \mathbb{E}\left[ c^1\left( t_{\text{max}}^{\pi(k)}, T_s^{\pi(k)} + \sum_{j=1}^{k-1} T_f^{\pi(j)} \right) \right] \right) \right)
$$

$$
\leq \sum_{k=1}^{N} \left( p_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( 1 - p_{\pi(i)} \right) \cdot \left( r_{\pi(k)} - c^1\left( t_{\text{max}}^{\pi(k)}, \mu_s(\pi(k)) + \sum_{j=1}^{k-1} \mu_f(\pi(j)) \right) \right) \right) = \hat{f}(\pi)
$$

(A.4)

The relaxed problem, $\hat{P}(A, t_{\text{max}})$ is identical to the problem given in equation 2.12 with the exception that the objective function of the relaxed problem is $\hat{f}(\pi)$ rather than $f(\pi)$. $f(\pi)$, $\hat{f}(\pi)$, are both evaluable on any permutation $\pi$ of integers $(1, \ldots, N)$ (where $A = \{1, \ldots, N\}$).
Appendix B

Proof of Theorem 2.2

B.1 Proof of Theorem 2.2

Proof. \( p_i > 0, 1 - p_i > 0, r_i > 0, \mu^f_i > 0, \mu^s_i > 0 \) \( \forall i \in (1, \ldots, N) \) thus:

\[
\hat{f}(\pi) \leq \sum_{k=1}^{N} \left( p_{\pi(k)} \cdot \prod_{i=1}^{k-1} \left( 1 - p_{\pi(i)} \right) \cdot \max \left( r_{\pi(k)} - c^1(\max(\mu^s_{\pi(k)}, \mu^f_{\pi(k)}), \sum_{j=1}^{k-1} \min(\mu^s_{\pi(j)}, \mu^f_{\pi(j)})) + 0 \right) \right) \tag{B.1}
\]

\[= \hat{\hat{f}}(\pi) \]

Where \( \hat{\hat{f}}(\pi) \) defines the objective function for our second relaxation.

The relaxed problem \( \hat{\hat{P}}(A, t^{max}) \) is then identical to the problem given in equation 2.12 with the exception that the objective function of the relaxed problem is \( \hat{\hat{f}}(\pi) \) rather than \( f(\pi) \). \( f(\pi), \hat{f}(\pi), \) and \( \hat{\hat{f}}(\pi) \) are all evaluable on any permutation \( \pi \) of integers \( (1, \ldots, N) \) (where \( A = \{1, \ldots, N\} \)). \( \square \)