March 2020

Optimization and Training of Generational Garbage Collectors

Nicholas Jacek

OPTIMIZATION AND TRAINING OF GENERATIONAL GARBAGE COLLECTORS

A Dissertation Presented

by

NICHOLAS JACEK

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2020

College of Information and Computer Sciences

OPTIMIZATION AND TRAINING OF GENERATIONAL GARBAGE COLLECTORS

A Dissertation Presented

by

NICHOLAS JACEK

Approved as to style and content by:

J. Eliot B. Moss, Chair

Benjamin Marlin, Member

Philip Thomas, Member

Antony Hosking, Member

James Allan, Department Head
College of Information and Computer Sciences

DEDICATION

For Christine, without whom none of this would have been possible.

ACKNOWLEDGMENTS

I begin by thanking my adviser, Eliot Moss, for his invaluable help and guidance. I also thank my committee — Tony Hosking, Ben Marlin, and Phil Thomas — for their thoughtful feedback which greatly improved this work. Special thanks are due to Phil who was a friend to whom I could turn to for advice long before he was a committee member.

Thank you to Sridhar Mahadevan, who was my adviser when I began graduate school, for beginning me along this journey and laying the foundation that eventually became this thesis. I further thank all the faculty and staff of the College of Information and Computer Sciences who have always provided a welcoming and engaging community.

I thank my siblings — Victor, Lucy, Steven, and Jenna — and my parents — Victor and Rae Ann — for their continual encouragement. Finally, I thank my wife, Christine, for her unflagging love and support.

ABSTRACT

OPTIMIZATION AND TRAINING OF GENERATIONAL GARBAGE COLLECTORS

FEBRUARY 2020

NICHOLAS JACEK, B.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

Garbage collectors are nearly ubiquitous in modern programming languages, and we want to minimize the cost they impose in terms of time and space. Generally, a collector waits until its space is full and then performs a collection to reclaim needed memory. However, this is not the only option; a collection could be performed early when some free space remains. For copying collectors, which are what we consider here, the system must traverse the graph of live objects and copy them, so the cost of a collection is proportional to the volume of objects that are live. Since this value fluctuates during a program's execution, a collector can minimize its cost by carefully choosing the points at which it collects.

We help to realize this goal in two ways. First, by developing an algorithm that analyzes after-the-fact traces and computes optimal collection points, we can explore the theoretical limits of garbage collectors. This gives insight into what performance gains are possible, and can guide future collector development into areas that could be most fruitful.

Second, we use techniques from machine learning to find improved garbage collection policies that could be implemented in real systems. The optimal collection points provide ground truth from which a model can learn.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer programs continually allocate data for future use. If every allocated objects is retained forever, eventually the computer's memory will completely fill, and the program will be forced to stop. Instead, once an object is no longer needed it should be freed so that the memory it occupies can be reused. This deallocation can be done by hand, as it is in C and C++, but the process is notoriously tedious and error prone. Most modern programming languages and run times take the alternative strategy of employing garbage collectors (GCs) – systems that automatically detect and remove unneeded data or garbage. These come with trade-offs, however. One study, Hertz and Berger 2005, found that in order to match the performance of carefully hand-crafted memory management, Java programs with typical GCs require heap sizes that are three to five times larger. At sizes smaller than that, the performance of the garbage-collected programs quickly degrade. If we agree to the widespread assumption that users will tolerate no more than ten percent of a program's execution time going to garbage collection, it is clear that the memory requirements of the GC can be considerable.

Many types of GCs exist in the literature and one might wish to optimize various different aspects of their operation based on the needs of a specific application. In this work, we focus on generational mark-compact collectors, similar to those used in Java virtual machines. Our analysis seeks to minimize the total time overhead they incur over the entire execution of a program.

While we want GCs to run as efficiently as possible, little beyond loose worst-case bounds is known about their theoretically optimal performance, and their behavior is generally based on simple heuristics. The main contributions of this work address these two issues. First, we give a dynamic programming algorithm that exactly computes the schedule of collector actions for a given program and input that has minimum cost. In many cases there is substantial room for improvement over the default behavior of the collector. Then, we present several different policy algorithms that use features available during the execution of a program. For some traces, they are unable to improve on the default algorithm, but for others they give substantial improvement. In a few cases, the learned policies obtain optimal performance.

As stated, this research will focus on generational garbage collectors. These belong to the class of tracing collectors, which means they assume any object reachable by following a sequence of pointers may be used in the future and must be retained. All other objects are unreachable, so their space may be reused. The generational collectors we study divide the heap space into two sections: a small young space (called a nursery by some authors), and a larger old space. Objects are initially allocated into the young space. In a full collection, the entire graph of objects in both spaces is walked and every object reached is marked as live. These objects are then compacted into the old space. During a young collection, the collector assumes that all objects in the old space are live and only walks the graph of objects in the young space for marking. The marked objects are then copied into the old space and the young space becomes empty.

The bulk of the work involved in the collections comes from traversing the object graph and copying the live objects. Thus, for our simplified model, we assume that the cost of a young collection is proportional to the volume of data that is promoted and that the cost of a full collection is proportional to the total volume of live objects at the time of the collection. The default policy is to wait until the spaces fill and perform a collection once the remaining free space is smaller than is needed for an allocation. However, it is also possible to collect when there is more space remaining, one hopes at a point when the cost of collection is lower.

The two tiered layout of a generational collector is motivated by the generational hypothesis (Ungar 1984), which states that most objects fall into one of two clear groups: those that are used only briefly and those that live for nearly the entire execution of the program. Ideally, most objects in the young space will be short-lived and therefore dead by the time a young collection occurs. In contrast, any object that reaches the old space is hopefully long-lived, so the assumption that most objects in the old space are live is safe. Frequent walks of the old-space object graph would be mostly wasted work.

In order to base our analysis on real programs, we have instrumented a number of executions of programs from the DaCapo benchmark suite on the Java Virtual Machine. The collected data include the size of each object allocated, the creation and modifications of the pointers used in the program, and each method call in the program's execution. The allocation and pointer data allow us to calculate the birth and death times of each object, and to simulate the performance of a garbage collector on the traces. The counts of object allocations, method calls, and branches taken represent features that a garbage collection system can instrument to make more accurate decisions.

At each point in time, a collector can choose to perform one of three actions: no collection, a young collection, or a full collection. Our overarching aim is to investigate how total collection cost can be minimized by careful selection of the points at which collections are performed. This work is divided into three broad sections toward this goal.

First, we develop a precise model of our collector. This is slightly idealized from the real world, but it allows for mathematical analysis and precise quantification of the costs of different actions.

Next, we study *schedules* of collections. An optimal schedule is a function from the true underlying *state* of the collector to the optimal action. By following a schedule, the garbage produced by a program can be collected at minimal cost. It is important to note that the collector's state can in general only be known after the fact. Intuitively, it might be overall cheapest to take an expensive action now in order to get even greater savings later. Thus, the state can implicitly include facts about the future. Nevertheless, we give a dynamic programming algorithm that can calculate exactly optimal schedules given after-the-fact traces of program executions.

This is the first work in the literature to give the exactly optimal cost to manage memory for a program using a realistic GC. Previously, it was impossible to know how much the performance of a GC could be improved, even in theory. Our dynamic program gives a lower bound on cost that we can compare various collection strategies against, and it can guide future research to focus on programs and situations where useful improvements are at least possible.

Finally, we turn our attention to collection *policies*. These are functions from *observations*, features of a program that can be measured at run time, to actions. We formulate the development of a policy as a classification problem, and attack it with several types of classifiers.

Before we continue, we wish to build a bit of intuition about the complexity that arises in the choice of optimal collection points. Consider an extremely simplified situation where the cost of performing a collection is periodic and the only constraint on the collector is that collections must be no more than one unit of allocation away from each other; this is illustrated in Figure 1.1. The blue curve represents the cost of a collection at that point. Then, the simplest policy, the default, is to spread the collections out as far as possible. In our example, this results in collections at each integer along the vertical axis, shown as green dots in the figure. Since the wavelength of the cost function is not an exact multiple of the time between collections, the collections have a variety of costs. They sum to 19.57 over the entire trace.

You may suspect that, due to the simple nature of the cost function and collector model, only limited and local information would be needed to select collection points more intelligently. One strategy is to collect at each local minimum of the cost function, where

Figure 1.1: Example trace, collected with the default policy

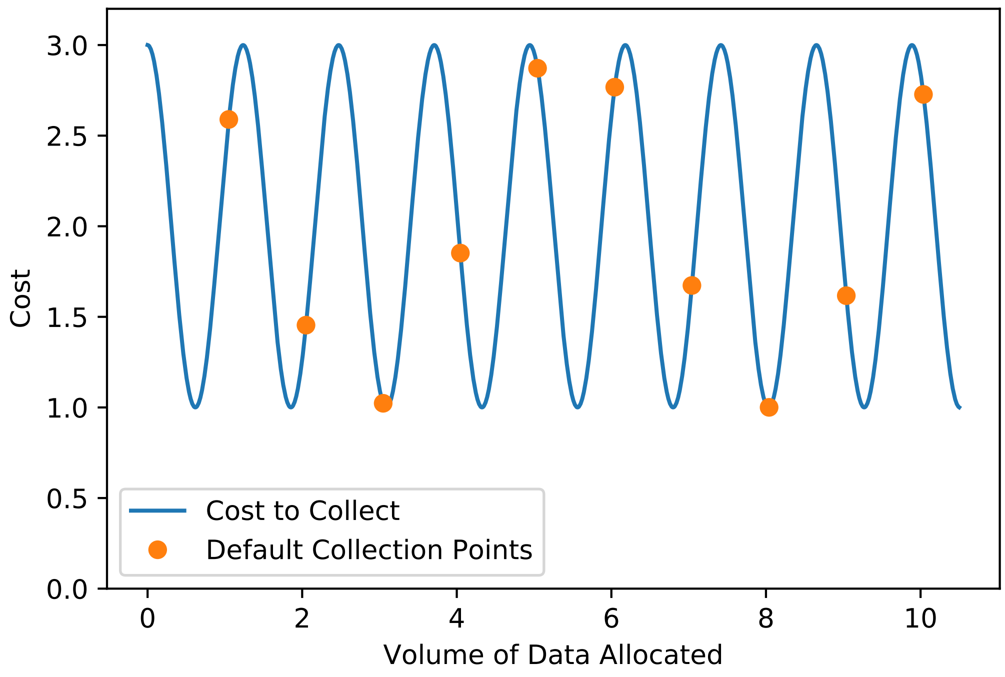
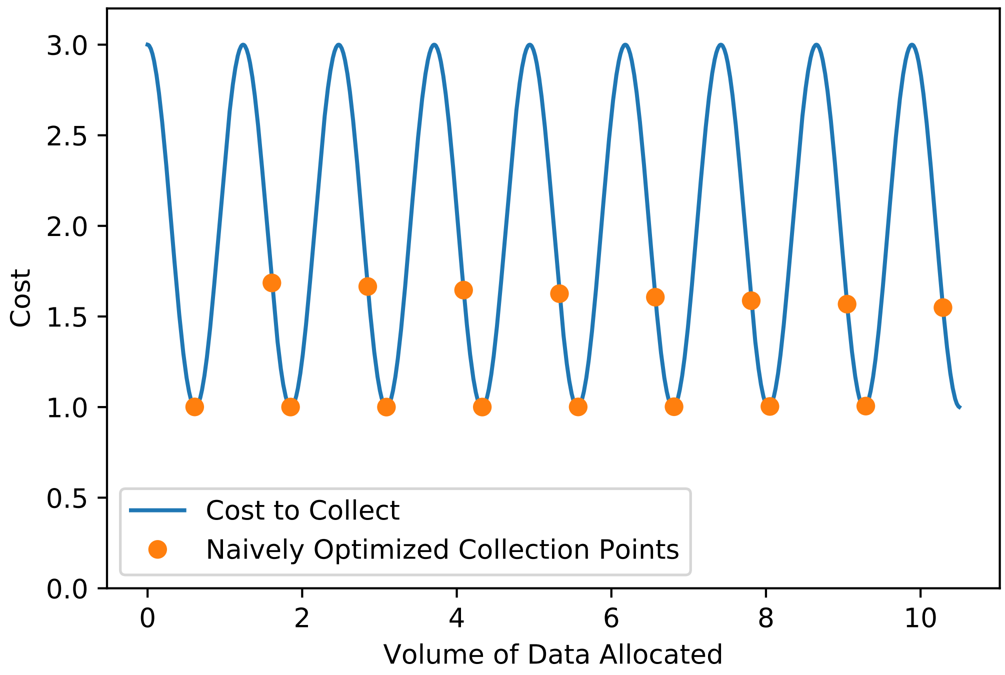


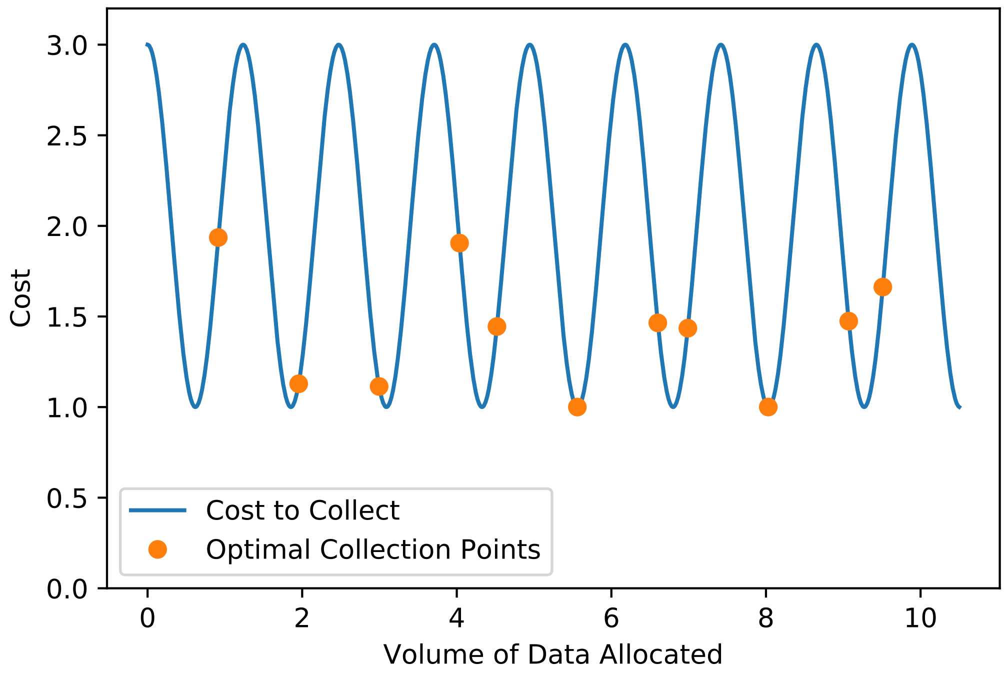
Figure 1.2: Example trace, collected with a naively optimized strategy



collections are least expensive. In a running system, the locations of these local minima may also be relatively easy to predict. The results of this naive optimization technique are shown in Figure 1.2. The wavelength of the cost function is slightly larger than the size of the heap. Thus, after one collection at a minimum, the collector is forced to collect again before it can reach the next trough. These extra collections are themselves costly, so the total cost of every collection in the trace is 20.94, more than the cost of the default. Simple decisions like this, even if they are locally optimal, are not always sophisticated enough to collect the garbage generated in an entire program efficiently.

In comparison, we show the exactly optimal collection locations in Figure 1.3. They were calculated using a dynamic program similar to the one we describe in Chapter 4, though greatly simplified for this much simpler model. All together, they sum to a cost of 15.56, showing that a significant cost savings is possible. Note, however, that the collection locations do not follow any obvious pattern. Though they tend not to be at the top of the

Figure 1.3: Example trace, collected optimally



cost curve, only two are located at minima. This illustrates an important property of these collection problems: choosing the optimal points requires *global* information about the trace. Whether we should collect right now to be optimal depends on what the program will do in the future.

Clearly, choosing optimal collection points in a real program is even more difficult. For example, we have to place both young and full collections, and the costs of young collections depend on what previous collection decisions we made. Furthermore, when choosing whether to collect at run time, we cannot exactly predict the cost of even a *single* collection, much less a series of them. Nevertheless, we have developed tools and techniques that can address these problems. To further set the stage for our work, in the next chapter we review the literature relevant to this research.

CHAPTER 2

LITERATURE REVIEW

2.1 Machine Learning for Garbage Collection

There have been several applications of machine learning to garbage collection in the literature. First, the most similar prior work comes from Andreasson, Hoffmann, and Lindholm (2002). The JRockit VM includes a concurrent garbage collector. Too frequent collections increase overhead, but if the heap fills completely a program must pause and wait for collection to finish. The authors use the reinforcement learning (RL) algorithm Sarsa to learn a policy that predicts the optimal times to begin garbage collection. However, the features used for learning are very simple and limited. In some experiments they use only the available space remaining in the heap, and in others they also include the change in available space. Nevertheless, they evaluate the learning algorithm using a synthetic program and hand-created cost model, and in some cases find increased performance over JRockit's default policy. The authors train and evaluate their system on the same program, and do not investigate whether the performance of their learned policy generalizes to different programs or heap configurations.

In this work, we use a richer set of features, allowing the representation of more sophisticated policies. Additionally, we evaluate the performance of the learned policies using traces collected from runs of a standard suite of benchmark applications. This allows for more realistic assessment of the impact the learned policies may have in actual practice. Finally, this research focuses on generational garbage collectors. At each time point, these have three possible actions: no collection, young collection, or full collection. This is more complicated than the decisions to collect or not explored in the cited paper.

While this may make policies more difficult to learn, there may also be more room to improve over default policies. Generational garbage collectors are more widespread in current run-time systems, which gives this research greater applicability to actual usage.

To our knowledge, the work of Andreasson, Hoffmann, and Lindholm (2002) is the only paper in the literature that attempts to learn policies that pick out points in time to perform garbage collections. So, it is the only research directly comparable to ours. Machine learning, however, has been directed towards the same overarching goal of improving the performance of garbage collectors by tuning other aspects of their operation.

One possible optimization is pretenuring, which is the allocation of new objects directly into the old space rather than the young space. If the object would be promoted eventually anyway, this saves work in a future young collection. Thus, accurate prediction of which objects are likely to be long-lived is necessary for efficient pretenuring. To this end, Singer, Brown, Luján, et al. 2007 investigated the information-theoretic concept of mutual information (MI) between various features of an object and its lifetime. Informally, MI measures how well we can predict one value given another. The authors report MI values for several object features, and find that a combination of an object's static allocation site and its dynamic calling context is the best predictor of object lifetime that they examine. But they do not develop any system that actually uses this information for pretenuring. In contrast, a main focus of this research is to learn policies and evaluate them on realistic models of garbage collection systems.

Another aspect of a garbage collector that can be tuned is its heap size. A too small size will increase the overhead from frequent collections, but a too big size will decrease memory locality and increase the occurrence of cache misses. In White et al. (2013), the authors use a proportional-integral-derivative (PID) controller to resize the JVM heap in order to keep garbage collection overhead at a user-specified target. The input to the PID controller is the time cost of the most recent GC divided by the time since the most recent GC, and at each collection it outputs a multiple by which to resize the heap. The PID is tuned for each benchmark using the standard Ziegler and Nichols method (1942). The authors find that the controller gives better performance than the heap resizing heuristics built into the Jikes and HotSpot JVMs, which tend to be too conservative in that they change the size of the heap too slowly. Our research instead tries to minimize collection cost given a fixed heap size. This allows for more efficient usage of memory, especially in cases where the total available memory is limited.

Next, the Memory Management Tool Kit for the Jikes RVM contains several different garbage collection schemes, such as semispace, mark-sweep, generational copying, etc. These schemes have different relative performance on different programs, so Singer, Brown, Watson, et al. (2007) used machine learning to predict the optimal collection scheme for a given program. They collect features from static analysis of a number of benchmark programs, as well as dynamic features from a single run of each program. Their ground-truth cost model is the total execution time of a program using a given collection scheme. The features are used to train a number of C4.5 decision trees that predict for a given input which of two schemes will have the lower cost. By arranging them in a tournament, the trees can then be used to predict the optimal collector for an input. Though one profiling run of a new program is still necessary, this is still much cheaper than exhaustive trials of all the available collection schemes.

Mao, Zhang, and Shen (2009) extend this work to predict the best collection scheme based on the input to a given program. They find that for a given heap size ratio (HSR), which is the ratio between the size of the heap used for a program execution and the minimum possible heap size for the program, one or two collection schemes are almost always optimal regardless of the input to the program. This gives rise to a two-step prediction scheme: first, predict the minimum possible heap size based on static features of the input and calculate the resulting HSR, then use the same features to choose between the two best collection schemes for the predicted HSR. The authors accomplish the first task with a regression tree and the second with either a classification tree or the nearest neighbors technique. In all cases, they achieve enough accuracy to increase performance on their benchmarks.

Singer, Kovoor, et al. (2011) then investigated the selection of a garbage collection scheme for programs that use a Java implementation of MapReduce, a popular framework for distributed computation. They study the case of using MapReduce on a single processor with multiple cores, rather than across a compute cluster. There are six different schemes examined in this study: serial, parallel, or concurrent collection, each using a young size to old size ratio of either 1:2 or 1:8. Static features of the JVM configuration and dynamic features from a profiling run of each program are collected and used to train binary classifiers. These are random forests of 20 trees, and predict whether a scheme is "good" for a program, meaning that the program will execute with this scheme in less that 95% the time used by the default scheme, and less than 110% the time used by the optimal scheme. The collector for a new scheme is then predicted by running it through the classifiers one by one in a fixed order, and selecting the first "good" scheme found. This improves performance compared to the default collector.

These three papers all focus on making very coarse decisions. Before a program is run, they choose from a handful of fixed collection schemes and then make no other decisions for the entire program's execution. Our work focuses on one scheme, generational collection, then analyzes and optimizes its performance. This gives lower cost in the cases for which generational collectors are already optimal, and may make generational collectors competitive with other schemes in a larger space of programs.

Finally, Tiwari and Vala (2017) examine stub-scion pairs (SSP), a method for reference-counting garbage collection for systems distributed over networks. They build a Bayesian belief network that models how likely different objects are to have pointers to each other. This information is then used to prioritize whether SSP chains should be created in advance, and which SSPs to prioritize for reclamation. They find that this can reduce the network overhead on a synthetic benchmark.

Our research focuses on tracing, rather than reference-counting collectors, and investigates the single-processor, rather than the distributed case. Thus, it is directed toward a very different goal than ours.

2.2 Feature Selection

When we look to build practical GC systems, each program presents thousands of possible features, far too many to ever be used in practice. We need to select a small group of them, but we need some way of choosing those that will be most useful in deciding whether to collect. The problem of finding the most useful subset of features from a large group is known as the feature selection problem, and in this section we give an overview of the extensive literature that has been produced on it.

Our presentation follows Guyon and Elisseeff (2003) in grouping feature selection algorithms into three broad groups: variable ranking methods, wrapper methods, and embedded methods. As we explain later in Section 5.2, we have used one method from each of these groups in order to give a wide sampling of the possible approaches.

In variable ranking methods, the first group, we compute some measure of a feature's usefulness and then simply select the features that score highest by this measure. Perhaps the simplest possible ranking criterion is the correlation between each feature and the output we are trying to predict. This can capture only linear relationships between the features, which is often not informative enough for difficult problems. However, Weston et al. (2003) find that this simple technique can give good results when used for gene microarray analysis, in which a few genes of interest must be located from among thousands of irrelevant ones.

Another possible ranking measure is the performance of single variable classifiers. That is, a classifier is restricted to use just a single variable, and trained to predict the output. The accuracy of the classifiers' prediction becomes the ranking criterion. Forman

(2003) applies Naive-Bayes classifiers to this technique, and finds it gives performance competitive with much more sophisticated methods for some problems.

Finally, information theory provides a range of measures that may be suitable for different applications. The mutual information between each feature and the prediction target is the most commonly used such measure. The algorithm of Fleuret (2004) extends this approach to include the conditional mutual information among the candidate feature and the target, given each previously selected feature. We apply this approach to our problem, so we defer detailed discussion of it to Section 5.2.3.

The next class of feature selection algorithms is wrapper methods. An early and informative study of these algorithms is given by Kohavi and John (1997). Selection of the exactly optimal subset of features in NP-hard (Amaldi and Kann 1998), so wrapper methods instead repeatedly make small changes to the set of selected features until a locally optimal set is found. At each step, classifiers are trained and tested to evaluate which modified set of selected features is best. One of the main benefits of these methods is that they can use the same classification algorithms to guide feature selection as will be used in the final classifier. This helps to give us confidence that the selected features will be genuinely useful. Many algorithms for modifying the selected feature set have been used. One of the simplest is the forward-stepwise technique. It begins with an initially empty set, and adds features one by one according to which most improve classifier performance. Similarly, the backward-stepwise begins by selecting all available features, and eliminates them one by one, guided by the performance of the classifier. In this class, we use the technique of random forest importance (Archer and Kimes 2008), since we are also using random forest classifiers. Detailed explanation of this technique is given in Section 5.2.2.

The final group of feature selection algorithms is embedded methods. These algorithms train classifiers and select the features that will be used as two parts of one unified whole. One example comes from decision trees (Breiman 2001), which select a feature and associated threshold for each node of the tree. The set of selected features is then simply the set of features that are used at any node. Another very popular class of embedded methods is to add a penalty, such as l_1 regularization, to the cost function that the classifier seeks to optimize. The penalty pushes many of the weights the classifier learns to zero, effectively de-selecting the features associated with these weights[1]. Bach et al. (2012) give a very thorough overview of these algorithms. As explained in Section 5.2.1, we use Group Orthogonal Matching Pursuit (Swirszcz, Abe, and Lozano 2009), which includes embedded feature selection.

2.3 Behavioral Cloning

Our learned policies fall into the broad category called learning from demonstration or imitation learning, in which the learner tries to copy the actions of a human or algorithmic expert. This basic idea has been explored from different directions and different disciplines, but one early and influential paper is that of Schaal (1997). In it the authors find that good policies for some problems can be learned much more quickly from demonstration than by standard reinforcement learning methods.

Many different aspects of a task, such as a model of the system to be controlled or a plan to execute, can be learned from expert demonstrations. We take perhaps the simplest approach where we learn a mapping from system states to actions to be taken. This strategy is sometimes called behavioral cloning, since we seek to clone exactly the expert's behavior. For systems such as ours where the agent must decide between taking discrete actions, the mapping takes the form of a classifier. In the literature, many different classifiers have been applied to many different control tasks, which we summarize below.

[1]In early work, we tried to apply these methods to our problem. We found that because we needed such a tiny ratio of selected to de-selected features, these algorithms tended to become numerically unstable and extremely difficult to tune.

First, both Baysian networks and k-nearest-neighbors classifiers have been applied to the problem of obstacle avoidance and navigation by Inoue, Inamura, and Inaba (1999) and Saunders, Nehaniv, and Dautenhahn (2006). Chernova and Veloso (2007) train Gaussian mixture models to control a car. Next, support vector machine classifiers (Cortes and Vapnik 1995) were applied to the task of robotic ball sorting (Chernova and Veloso 2008). Most similar to the techniques we employ, Sammut et al. (1992) train decision trees to control simulated airplanes. Finally, Stéphane Ross, Gordon, and D. Bagnell (2011) and (Stephane Ross and J. A. Bagnell 2014) relatively recently developed algorithms that intelligently query their experts for specific demonstrations in order to improve the performance of their learned policies, which are represented by neural networks.

CHAPTER 3
GENERATIONAL GARBAGE COLLECTOR MODEL

In this chapter we describe our model of a generational collector. We begin with an intuitive explanation of our model, then continue with a description of the traces we collect in order to simulate the behavior of our collector on Java programs. Finally, we give a precise mathematical description of a generational garbage collector.

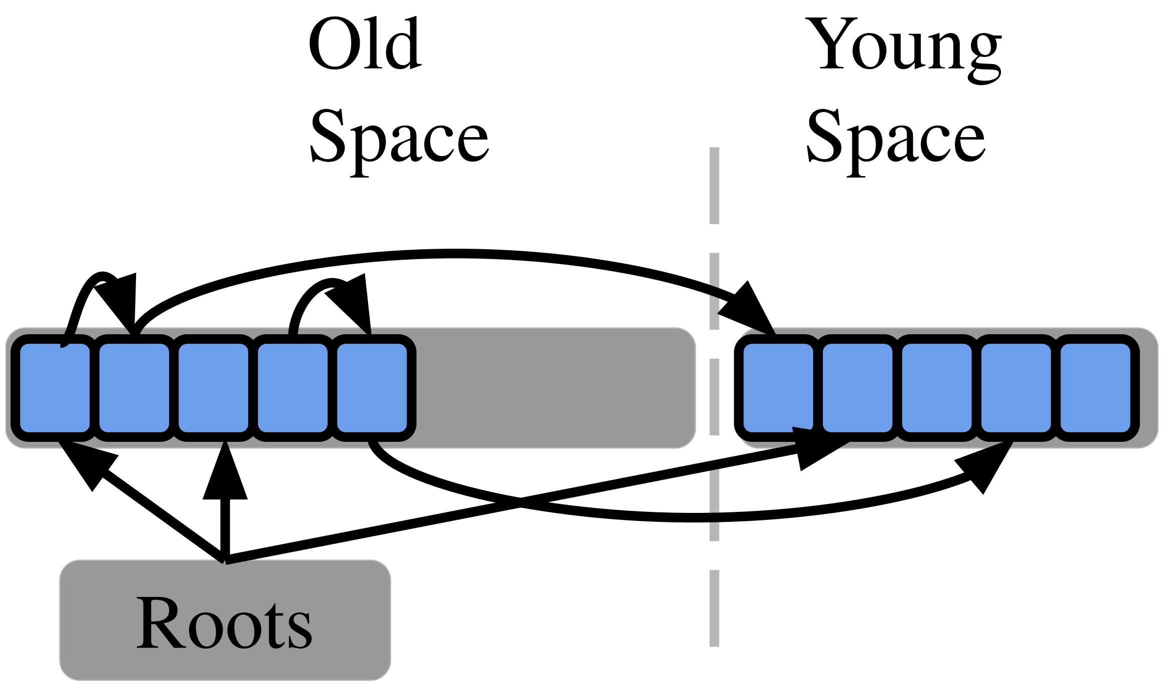
3.1 Collector Model

We are concerned with a *generational* garbage collector. In our model, the heap is divided into two spaces: the large *old space*, and the smaller *young space*. This is illustrated in Figure 3.1. Objects are allocated into the young space, unless they are larger than the young space itself. In this case, a collection must occur to empty the young space, then the object is allocated into the old space instead. The collector can perform two types of collections. A *full collection* reclaims all available space and a *young collection* frees space only in the young space. In the remainder of this section, we give intuitive explanations of these two types of collections based on the behavior of the program and run-time system. Later, in Section 3.3, we abstract away these considerations into a mathematical model of the collector.

3.1.1 Full Collections

The simpler type of collection is the full collection, shown in Figure 3.2. Any object that is reachable by following a chain of pointers from the program's local and global variables may be used in the future and must be retained, so we consider them *live*. All

Figure 3.1: Generational Collector Model



This is a graphical representation of our collector model, showing how it is divided into a young and an old space. The blue boxes are objects, and the arrows show the pointers between them.

other objects cannot be reached and will never again be used by the program. We consider them *dead*, and it is safe to free the heap space they occupy and use it for new allocations. The first step in a full collection is therefore to locate all live objects. The collector begins with the local and global variables, collectively called roots. Then, the collector simply walks the graph of pointers and objects and marks each object it encounters as live. Any object not marked must be dead. Finally, the marked objects are compacted into the old space, leaving the some of the old space and the entire young space empty. If instead the total size of the live objects is larger than the old space, the data cannot be collected and the program terminates.

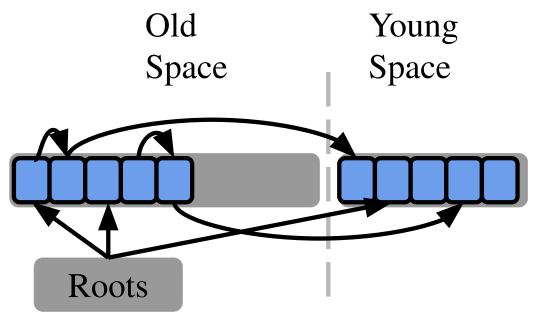
3.1.2 Young Collections

In contrast to full collections, young collections empty the young space but do not reclaim dead objects in the old space. They are shown in Figure 3.3. Instead of starting from the roots, during a young collection the collector begins marking from any pointer into the young space. This is equivalent to assuming that all objects in the old space are live. Of course, this is not always true and some objects in the young space will be marked even though they are reachable only from dead objects in the old space. We call these objects *baggage*. After marking, the marked objects are *promoted*, or copied into the old space, leaving the young space empty and ready to receive more allocations.

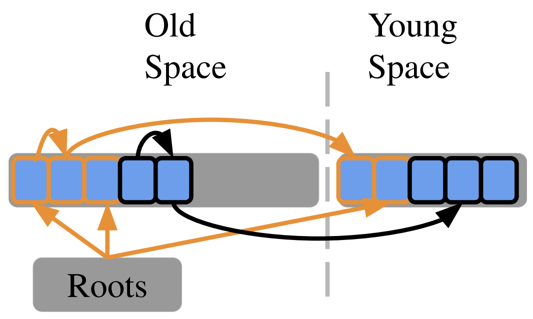
A young collection is generally cheaper than a full collection because the volume of objects examined is lower—a full collection traces through all reachable objects, and the young space is smaller than the old space. The two-tiered design of a generational collector is further motivated by the generational hypothesis. This states that objects will either die quickly or live nearly forever. Ideally, short-lived objects will die before a young collection occurs and only long-lived objects will be promoted. Then, nearly all the objects in the old space will be live so the overhead of needlessly promoting baggage will be minimal.

Figure 3.2: Full Collection Model

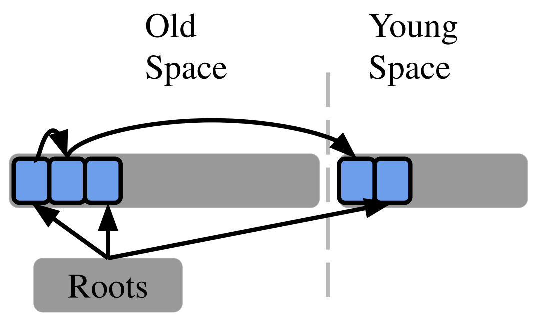
(a) Initial configuration.



(b) Marking phase of a full collection. Beginning with the roots, all reachable objects are marked. The followed pointers and marked objects are highlighted in orange.



(c) After marking, any unmarked object is considered dead. The remaining objects are all live.



(d) Finally, the live objects are all compacted into the old space. Now, the young space is empty and ready to receive new allocations.

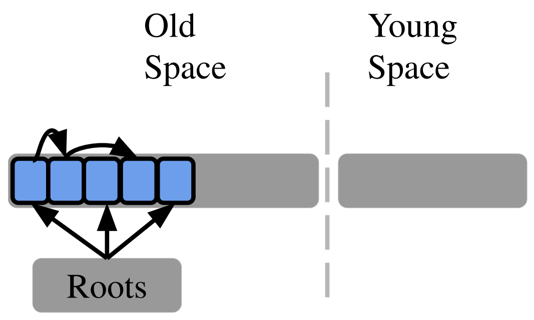
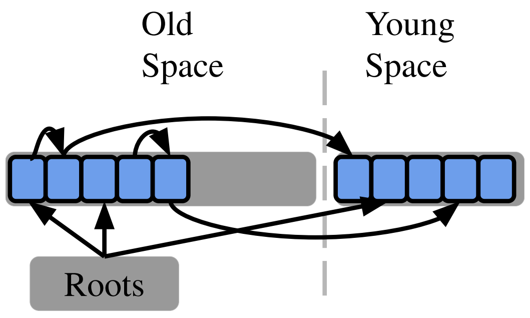
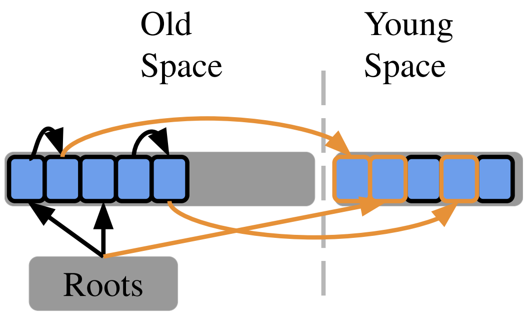


Figure 3.3: Young Collection Model

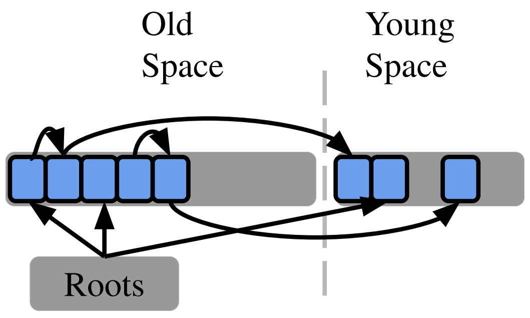
(a) Initial configuration.



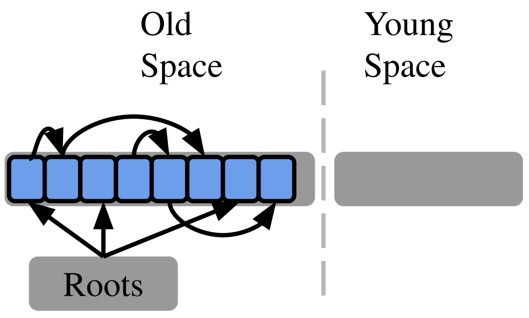
(b) Marking phase of a young collection. The collector follows all pointers into the young space and marks the objects found, even if the pointers originate from dead objects in the old space. The marked objects and followed pointers are highlighted in orange.



(c) After marking, any unmarked object in the young space is considered dead. The collector conservatively assumes that any marked object may still be reachable.



(d) Finally, the marked objects in the young space are all copied into the old space. Some copied objects are genuinely live, and others are promoted as baggage. Again, the young space is empty and ready to receive new allocations.



3.1.3 Cost Model

It is known that the running time of collectors that copy objects—the kind we consider here—is roughly proportional to the volume of objects copied, so we use the number of bytes copied as our cost model. Therefore, the cost of a young collection is the volume of objects copied from the young to the old space, and the cost of an old collection is the size of all the reachable objects. Other cost models are possible, such as charging a "rental" for volume of data in the heap at each time step. Our work focuses on minimizing total garbage collector effort, rather than trying to reclaim heap space as quickly as possible

3.1.4 Alternative Models

Though generational collectors are popular, other types have been developed and studied. One well-known type is the *copying* or *semi-space* collector. In this setup, the heap is divided into two equally-sized halves, one of which is always empty during allocation. New objects are created in the partially filled space until it is full, at which point a collection is triggered. In the marking phase of a collection, the entire object graph is traversed, starting with the roots. The marked objects are then copied into the other empty space, leaving the originally filled space empty. In a copying collector, the cost of a collection comes from traversing the object graph and copying objects, as in our generational model. Since there is also only one type of collection, it should be possible to analyze and optimize a copying collector with techniques very similar to those we develop. However, we believe the higher cost for each collection and lower space utilization will give less opportunity for optimization.

In contrast, a *mark/sweep* collector works quite differently to those we study. In these systems, after a marking phase, objects are left at whatever positions the happen to occupy. Instead, the collector manages a free list which records the non-contiguous locations in memory that can receive allocations. Object allocations are then much more complicated as appropriately sized chunks of free memory must be found on the free list, and the free list must be updated and maintained. A particular problem called fragmentation can occur when only many small chucks of memory are free. Though the overall fraction of the heap that is in use may be relatively low, it may still be impossible to find space to allocate a large object.

Since allocation as well as collection can incur significant costs in a mark/sweep collector and collections do not involve copying objects in the heap, the cost model for these collectors is completely different from those that we study. Therefore, our results are not likely to be directly relevant to them.

Finally, collectors have been developed that include more than two generations and correspondingly increased types of collections. In our work (Jacek, Chiu, B. M. Marlin, et al. 2019), we note that our dynamic program can be extended to these models in a relatively straightforward manner. Unfortunately, adding additional generations greatly increases the asymptotic time complexity of our algorithm, making it infeasible to run on program traces of realistic length. We were thus unable to pursue this line of research empirically.

3.2 Nature of Our Traces

We used the Elephant Tracks (ET) tool (Ricci, Guyer, and Moss 2011; Ricci, Guyer, and Moss 2013) to obtain sequences of event records from executions of Java programs. There are two kinds of relevant events in these traces: control events (method calls, returns, etc.) and heap events (object allocation, pointer updates, object death). Like its intellectual predecessor Merlin(Hertz, Blackburn, et al. 2006), ET computes precise death times for each object, i.e., the point at which the object was last reachable. ET's strategy for this is to record when references to heap objects exist. ET then determines death time as the last time an object was reachable from a root.

We post-process these event traces in two significant ways:

1. By simulating the heap events (allocations, pointer updates, and deaths), we can determine the pointers within each object when it dies. We then use that information to compute the *pre-birth* time for each object, a concept introduced by Jacek, Chiu, B. M. Marlin, et al. (2019). Consider conceptually a heap large enough to hold all objects allocated during a program's execution. Suppose we inspect that heap at the end of the run and determine, for a given object *o*, the set of objects from which we can reach *o* in the heap, i.e., the predecessors of *o* in the heap graph. (The heap graph is the directed graph where objects are nodes and pointers are the directed edges.) The pre-birth time of *o* is the minimum (earliest) of the birth (allocation) times of *o*'s predecessors.

   Knowing the pre-birth time is significant because it enables direct determination of whether a given young collection will preserve *o*. Of course *o* will be preserved if it is *live* (reachable) at the time of the collection—a requirement of garbage collector correctness. However, if the previous collection (young or full) occurred between the pre-birth and birth time of *o*, and the next collection is a young collection after *o* dies, *o* will also be preserved. (In the terminology of Jacek, Chiu, B. M. Marlin, et al. (2019), this is the case in which *o* is *baggage*.)

2. We *group* events of the trace. Every 256 Kbytes of allocation forms an allocation group, and we also group the control events that occur in the same interval. Specifically, if adding a second or later object to the current group would cause the group's size (bytes allocated) to exceed 256 Kbytes, then the allocation event of that next object starts a new group. Groups are reasonable in that most real allocators will make a decision about whether to run the garbage collector only as a block of some size fills. Groups serve several purposes in this work.

- Groups facilitate calculating optimal collection schedules efficiently, as discussed by Jacek, Chiu, B. M. Marlin, et al. (2019), because we can deal with whole sets of objects at once rather than handling each object separately.
- Pre-computed groups further insure consistent definition of the possible times for collection across heap sizes and previous collection histories.
- Groups are our basis for defining feature vectors from the control events of a trace. In particular, within a group we compute for each Java method two features concerning calls of that method: the number of times the method was called, and a 0/1 feature that indicates just whether the method was called at all. Each source of control events determines a similar pair of features: calls of a method, returns from the method, exception throws and catches, calls from a given call site, and allocations at a given allocation site. We also have features for the number of bytes allocated at an allocation site, and the number of array elements allocated at a site that allocates arrays.

  A given program may have tens of thousands of possible features. Typically only about 10% of the possible features are actually used in a given execution, and of course many of those are zero in the time window of a given group. Still, the number of features is large, so ultimately it is important to control how many are used in a learned policy function. This is true both because obtaining a feature's value at run time has a cost every time the feature's event occurs, and because evaluating the learned function will be costly if it uses a large number of features.

3.2.1 Trace Products

There are two key post-processed products for each trace:

- Allocation cohorts: A *cohort* is a set of objects whose pre-birth times fall into the same group, whose birth times fall into the same group, and whose death times fall into the same group. Because those times entirely determine the collector's behavior and costs (under our models), for modeling collection behavior all we need to know is the set of cohorts and their sizes.

- Feature vectors: Concerning the feature vectors, feature numbers for the same method may vary from trace to trace, since the overall set of methods can be different. When we handle multiple traces from the same program, we first map all the features of the individual traces onto the union of features across the traces.

An additional trace product is the number of times instrumentation would be triggered for each feature, i.e., an estimate of the relative cost to obtain that feature at run time. (At present we do not exploit this information.)

3.2.2 Trace Details

Table 3.1 lists the programs from which we gathered traces, indicating the number of traces for each program and the ranges of number of groups, number of cohorts, bytes allocated, and maximum live size (maximum number of bytes reachable at once) for the traces of that program. Except for javac, these are all from the DaCapo benchmark suite (Blackburn et al. 2006), though we developed additional inputs for most of them. In the case of javac, the program is a modified version of the original SPECjvm benchmark of the same name, but modified to avoid caching of class file information across compilation of multiple classes, to simulate better what a compilation server might be like. Across the programs there is considerable variation in the statistics, and for many of the programs considerable variation across traces.

3.3 Mathematical Definition of a Generational Garbage Collector

Here, we give equations that define the behavior of our model of a generational garbage collector. As previously mentioned, we accumulate object allocations into groups of 256 kilobytes, and allow collections only at the borders between groups. Indices for indexed

Table 3.1: Summary of traces used

Program	Traces	Groups	Cohorts	Alloc (MB)	Max Live (MB)
avrora	4	181– 1606	944– 19603	45– 400	3.6– 105.7
batik	11	99– 586	586– 4766	25– 186	5.0– 24.6
fop	12	124– 6560	1024– 47124	30– 1587	5.0– 38.3
javac	4	1860–15759	25410–308523	459– 3899	10.3– 16.5
luindex	5	26– 27	170– 183	6– 6	2.0– 2.0
lusearch	3	2744–43921	9936–155178	664– 10510	2.0– 2.2
pmd	19	53– 5043	358–134539	13– 1202	1.7– 168.0

values in our dynamic programs are thus *group numbers.* The group size (sometimes called the block size) is notated G. We use n as the number of groups in a given trace (for a given G). Grouping defines $n + 1$ positions in the trace (on either side of the n groups, like fenceposts and a fence), which we number 0 through n and refer to as *time steps.* In general, the notion of time we use is the number of bytes allocated so far in the trace. If A is the total number of bytes allocated, then $n < 2A/G$. (The factor of two come from the worst case grouping where each object is just over $\frac{1}{2}G$ in size.)

We can now state our task more precisely. At each time step, our collector must perform one of three actions: no collection, a young collection, or a full collection. Note that although we define $n + 1$ time steps, the collector only has to choose $n - 1$ actions in an entire trace. At time 0 no allocations have yet happened, and at time n the trace has ended and no objects are live. The inclusion of these time steps makes our notation simpler and more consistent, but at these points all three actions have identical effects and zero cost. Also, not every action is always valid. For example, if the old space is full, the collector cannot successfully execute a young collection.

In order to determine which actions are valid and track their effects, we must precisely express the states that the collector can be in. For this, three numbers suffice:

1. t - The current time step.

2. l - The time step of the last collection the collector preformed. This is needed to calculate which objects will be promoted as baggage during a young collection.

3. u - The volume of data in the old space. This allows us to determine whether a young collection should be disallowed because the data it may promote could overflow the heap.

We now give some preliminary definitions to prepare for explaining the effect of the three actions. Every object x has a pre-birth time $p(x)$, a birth time $b(x)$, and death time $d(x)$. Necessarily $p(x) \leq b(x) \leq d(x)$. It is helpful to think of the given times as picking out instants, and the events as occurring between these instants. So, if for some object x we have $p(x) = b(x) = d(x) = t$, we know that the object was pre-born, born, and died at some times between the instants picked out at t and $t+1$. This helps clarify what happens when we consider a garbage collection to happen "at" time t—for a full collection, objects x with $b(x) < t \leq d(x)$ are live and will be retained. Others are either dead ($d(x) < t$) and will be reclaimed or have not yet been allocated ($t \leq b(x)$).

The p, b, and d of each object are mapped to the group they fall within, the groups being numbered 0 through $n-1$. Objects with the same mapped (p, b, d) triple exhibit the same collection behavior and form a cohort. We use $V[\ldots]$ to denote subscripting of the various arrays of our dynamic programs. Then, $c[p, b, d]$ is the size of the cohort with pre-birth group p, birth group b, and death group d, i.e., the sum of the sizes of the objects in the cohort. Here $p \leq b \leq d$. The values of $c[\cdot,\cdot,\cdot]$ completely capture the behavior of the trace for the given group size, and form the basis of the optimization problem to be solved.

Given $c[\cdot,\cdot,\cdot]$ it is easy to define the live size, or volume of data that is live at a given point:

$$L[t] = \sum_{b<t\leq d} c[p, b, d]. \tag{3.1}$$

No data are live before the trace begins and after it ends, so we have $L[0] = L[n] = 0$.

Similarly, we define $A[i, j]$ to be the volume of data allocated between i and j. It is calculated as

$$A[i, j] = \sum_{i \le b < j} c[p, b, d]. \tag{3.2}$$

Finally, we use $S^Y$and $S^O$to denote the sizes of the young and old spaces, respectively. With this, we are ready to turn to the actions themselves. We give formulae for three aspects of each action. The first is the precondition. If in a certain state this is not met, the corresponding action cannot be taken. Next, we list the effect, which tells how each action will change the state of the collector. Finally, we give the cost of each action. We aim to minimize the sum of the costs of every action taken over a trace.

3.3.1 No Collection

We explain the simplest action first. Minimal bookkeeping is needed when no collection is performed.

- Precondition: $A[l, t + 1] < S^Y$ - Some collection previously emptied the young space at time step l, and we can only allocate into it until it fills. If the allocations in the next time step would overflow the space, we must perform some collection, and this action is not valid.
- Effect: $(t, l, u) \to (t + 1, l, u)$ - When no collection is done, the time step simply advances.
- Cost: 0 - If we do not collect, we incur no cost.

3.3.2 Young Collection

Next, we examine young collections. For clarity, in this subsection we let y denote the total volume of data that will be promoted during a young collection. We calculate it as a sum of the affected cohorts:

$$y = \begin{array}{lll} \sum_{l \le b < t \,\wedge\, t \le d} & c[p,b,d] & \text{(live)} \\ + \;\; \sum_{l < t \,\wedge\, l \le b \,\wedge\, d < t} & c[p,b,d] & \text{(baggage),} \end{array}$$

- Precondition: $u + S^Y < S^O$ - We cannot know in advance how much of the data in the young space will be promoted during a young collection before it is performed. We therefore take a conservative approach and allow a young collection only if the data already in the old space plus the entire contents of the young space would fit into the old space.

- Effect: $(t, l, u) \longrightarrow (t+1, t, u+y)$ - A young collection has three effects. We advance by one time step, record that a collection has taken place at step t, and add the volume of promoted data to the old space.

- Cost: y - The cost of a young collection is equal to the volume of data that is promoted.

3.3.3 Full Collection

We now look at the final of the three actions, the full collection.

- Precondition: $L[t] < S^O$ - After a full collection, all live data will be in the old space. We must ensure that it is large enough for this to happen.

- Effect: $(t, l, u) \longrightarrow (t+1, t, L[t])$ - As in a young collection, we advance to the next time step and record that a collection has taken place. In contrast, the occupancy of the old space is equal to the live size after the collection.

- Cost: $L[t]$ - All live objects are compacted into the old space, so the cost of a full collection is equal to the live size.

Now that we have a complete definition of the dynamics of our collector model, we turn our attention to an algorithm to find the minimum cost of a schedule of collections for a trace.

CHAPTER 4

DYNAMIC PROGRAMS FOR FINDING OPTIMAL SCHEDULES

4.1 Introduction

We have developed and analyzed a realistic model of a garbage collector. Previously, we published an article (Jacek, Chiu, B. Marlin, et al. 2016) in which we used reinforcement learning techniques to find approximately optimal collection schedules. However, we directly improved on this work by developing a dynamic programming algorithm that finds exactly optimal collection schedules. The dynamic programming algorithm has obviated the earlier work, so we present only the improved algorithm here. We emphasize that this is the first work in the literature that calculates exactly optimal performance for a realistic garbage collector.

We develop the dynamic program in two large stages. The first stage considers only young collections. It addresses the constraint of young space size, but ignores the size of the old space. It develops arrays $y[i, j]$, $Y[i, j]$, and $\bar{Y}[i, j]$. The second stage introduces full collections and considers their optimal placement given the model of optimal young collection placement. It develops arrays $f[i, j]$ and $F[t]$. This two-stage development ignores the consideration of objects that meet or exceed the size of young space, so we add a refinement that deals with that. It develops an array $B[t]$ (for "big").

4.2 Dynamic Program

The algorithm proceeds by filling in a series of dynamic programming tables each of which builds on the last. We now describe these tables in turn.

First, consider the cohorts that are promoted[1] during a young collection. They fall into two categories. The first is the live objects that have not yet been promoted, that is, objects that were born after the previous collection but have not yet died. The second is *baggage*, the objects that were pre-born during the previous collection and have since died. All together this implies that, in order to calculate the cost of a young collection and the volume of data that it promotes, we need only one piece of information in addition to the properties of the cohorts themselves: the time step of the previous collection. This allows us to define $y[i, j]$, which is the cost of a young collection at time step j, given that the previous collection was at step i. (Necessarily $i < j$.) For convenience, we adopt the convention that $y[i, j]$ is infinite if the volume of data allocated between i and j is larger than our young space size, S^Y. Formally, we calculate $y[i, j]$ as

$$y[i,j] = \begin{cases} \quad \sum_{i \le b < j \,\wedge\, j \le d} \quad c[p,b,d] \quad \text{(live)} & \\ + \quad \sum_{p<i \,\wedge\, i \le b \,\wedge\, d<j} \quad c[p,b,d] \quad \text{(baggage)}, & \text{if } A[i,j] \le S^Y \\ \infty & \text{otherwise} \end{cases} \tag{4.1}$$

Again, $y[i, j]$ is the cost of a young collection at time j assuming the previous collection (young or full) was at i, i.e., there is no collection in between. (This intentionally does not include the cost of the collection at i, only that of the collection at j.)

Next, we widen our focus to schedules of collections. Let $Y[i, j]$ be the minimum cost of *any* schedule that (a) begins with a collection (young or full) at i, (b) ends with a young collection at j, and (c) has zero or more *young* collections in between (but no full collections). Note that because the cost is the volume of data promoted, the same schedule minimizes bytes traced and bytes promoted. This is important in determining impact on occupancy in the old space. As with $y[i, j]$, $Y[i, j]$ includes the cost of collecting at j but

[1]Observe that our cost model views promotion as essentially implying copying, so we tend to use the terms synonymously.

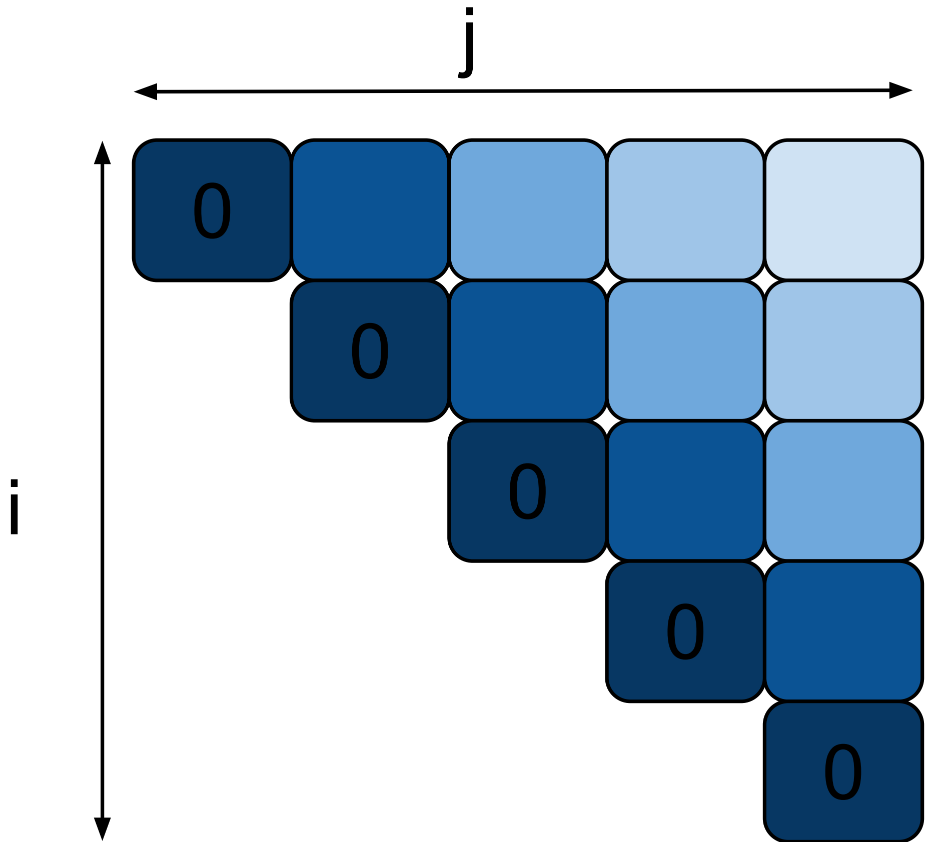
not the cost of collecting at i. Collecting a schedule of zero length has no cost, so we begin with $Y[i,i] = 0$. Then, we can calculate $Y[i,j]$ recursively:

$$Y[i,j] = \begin{cases} \min_{i \le k < j} Y[i,k] + y[k,j] & \text{if } L[i] + Y[i,k] + S^Y \le S^O \\ \infty & \text{otherwise} \end{cases}$$

There are two possible cases. First, there may be no young collections between i and j, giving cost $y[i,j]$. (Recall that if that is not feasible, $y[i,j] = \infty$.) This occurs in the equation when $i = k$, which implies that $Y[i,k] = Y[i,i] = 0$. We must also guarantee that the promoted data will fit into the old space. After a full collection at i, the amount of data in the old space will be $L[i]$, and $Y[i,k]$ data will have already been promoted from prior young collections. In practice, we do not know how much data will be promoted from a young collection beforehand, so we conservatively estimate that everything in the space, up to S^Y,may be promoted. Because we allow a young collection only if the sum of these values is less than the size of the old space, S^O,we guarantee that any young collection that is begun will be able to complete successfully. Second, there could be one or more young collections between i and j. In this case, let k be the location of the last such collection. Then, we can split the entire range into two smaller ranges, one from i to k and another from k to j. We have met the constraint that there is a collection at the end of each range, so the total cost of the schedule from i to j becomes the sum of the costs of the shorter segments: $Y[i,k] + y[k,j]$. Of course, there may be many possible ways to split the range into two segments. We choose the one that has the lowest cost. It is important in our later analysis that the second component of this sum is y as opposed to Y.

In practice, we calculate Y in a dynamic programming table, as shown in Figure 4.1. The value of a cell for a certain range depends only on the values for shorter ranges. So we initialize values along the diagonal to be 0, and can then work across each row (more generally, upwards from the diagonal).

Figure 4.1: Schematic view of the dynamic programming table. The values on the main diagonal can be initialized to zero. Then, the value in each cell depends only on the values in cells that have darker colors. Computation therefore proceeds upward and to the right.



Next, it is useful to relax the restriction that there must be a young collection exactly at the end of a certain range. This yields $\bar{Y}[i, j]$, which represents the minimum cost of a schedule from i to j that allows only young collections. Intuitively, we begin at each cell of $Y[i, j]$ and look backwards for the minimum cost schedule that ends within one young space size of j. The calculation of $\bar{Y}$ is straightforward:

$$\bar{Y}[i, j] = \min_{\{j' \mid i \leq j' \leq j \ \wedge \ A[j', j] \leq S^Y\}} Y[i, j']. \tag{4.2}$$

Finally, we calculate the costs of schedules that include full collections. We define $f[i, j]$ analogously to $y[i, j]$: it is the cost of a schedule that begins with a full collection at i and ends with a full collection at j, but has no full collections between i and j. It does, however, include the costs of any young collections between i and j. As with y, etc., it includes the cost of the collection at j, but not of the one at i. We further ensure that the amount of data promoted fits into the old space. That constraint has to do with the space in use at i, namely $L[i]$, plus the volume promoted during i to j.

$$f[i, j] = \begin{cases} \bar{Y}[i, j] + L[j] & \text{if } L[i] + \bar{Y}[i, j] \leq S^O \\ \infty & \text{otherwise} \end{cases} \tag{4.3}$$

At last we are ready to calculate the optimal schedules with arbitrarily placed young and old collections. We let $F[j]$ be the minimum cost of a schedule that begins at 0 and ends with a full collection at j. This is analogous to the first row of $Y[i, j]$. It would be straightforward to extend F to be a two-dimensional table, but the extra data are not needed for our purposes and would increase the cost of solving the problem. $F[0] = 0$, since there is no cost to collect a schedule of length 0. The calculation proceeds almost identically to $Y[0, \cdot]$:

$$F[j] = \min_{0<k<j} F[k] + f[k, j] \tag{4.4}$$

As before, we consider the cases where there are no full collections between 0 and j, and those where there are one or more. We take the minimum cost of all the various options. We again calculate these values using a dynamic programming table.

After the end of the trace, all objects have died, so there is no cost to collecting at n. This means that the minimum cost of a schedule for an entire trace can be read from the table: it is $F[n]$.

There is one additional complication we have glossed over. Objects larger than the young space are allocated directly into the old space, provided that the young space is empty at the time. Handling this condition requires some extra bookkeeping. We let $B[i]$ be the size of the group allocated at i, if it is at least as large as the young space size, and 0 otherwise. That is,

$$B[i] = \begin{cases} A[i, i+1], & \text{if } A[i, i+1] \geq S^Y \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

Then, we make the following modifications. First, the calculation of y becomes

$$y[i,j] = \begin{cases} \quad (\sum_{i \leq b < j < d} \quad c[p,b,d]) - \boxed{B[i]} \quad \text{(live)} & \\ + \quad \sum_{p < i \leq b \leq d \leq j} \quad c[p,b,d] \qquad \text{(baggage)}, & \text{if } A[i,j] - \boxed{B[i]} \leq S^Y \\ \infty & \text{otherwise} \end{cases} \tag{4.6}$$

In the calculation of the constraint, we subtract the size of a large object only if it occurs at i, which ensures that allocation directly into the old space is allowed only immediately after a young collection. Note that since these objects are allocated directly to old space, there is no cost to promoting them (hence the subtraction of $B[i]$). The differences from before are highlighted with framed boxes. Here a large object allocation (at i) is followed by zero or more allocations that fit into young space.

Next, we must ensure that every sequence of young collections has enough room left in the old space for each collection to complete successfully, even after taking into account the extra data allocated directly into the old space. This is accomplished by adding an extra constraint onto the calculation of Y:

$$Y[i,j] = \begin{cases} \min_{i \le k < j} Y[i,k] + y[k,j] & \text{if } L[i] + Y[i,k] + S^Y + \boxed{\sum_{i \le h \le j} B[h]} \le S^O \\ \infty & \text{otherwise} \end{cases}$$

.

Finally, we modify f:

$$f[i,j] = \begin{cases} \bar{Y}[i,j] + L[j], & \text{if } L[i] + \bar{Y}[i,j] + \boxed{\sum_{i \le k < j} B[k]} \le S^O \\ \infty & \text{otherwise} \end{cases} \tag{4.7}$$

. This adds the requirement that the promoted data still fits into the old space once the data allocated directly into the old space are included.

The dynamic program finds the minimum schedule *cost*, i.e., the minimum cost attained by any legal schedule. More than one schedule may have that cost, of course, but in any case, how can one extract the *schedule* from the solution, as opposed to just the cost? This is straightforward, and can be done after the fact or as we go. The "as we go" form is easier to explain: As we compute each table entry that has multiple options, we record also which option (which often includes an array index) we used for the minimum cost. Doing this along the way adds a constant factor overhead in space and time. Doing it after the fact requires some scanning, but it is proportional to n. For example, to find how $Y[i,j]$ was minimized, we scan values of k seeking to minimize $Y[i,k] + y[k,j]$. This is best done starting from $j-1$ and working backwards, stopping either at i or when $y[k,j]$ becomes ∞.

4.3 Calculation of the Action-Value Function for Arbitrary States and Actions

We have presented an algorithm that calculates the minimum cost of a schedule for an entire trace. Additionally, it is useful to be able to calculate the cost of beginning in an arbitrary state, taking an arbitrary action, and acting optimally thereafter to the end of the trace. In this section, we present an extension of our algorithm to this case.

First, we note that any state a collector is in can be completely characterized by three numbers: the current time step, i, the time step of the last collection of any type, l, and the volume of data in the old space, u. One might expect we would also need to know the volume of data in the young spaces as well, but since we know how much is allocated at each time step, this can be calculated from i and l.

Next, we need to augment F into a two-dimensional table that gives us segments of optimal schedules that begin and end at different points.

$$F[i,j] = \min \begin{cases} f[i,j], \\ \min\limits_{i<k<j} F[i,k] + f[k,j] \end{cases} \tag{4.8}$$

As with Y before, the F table can be filled in beginning at the diagonal and working upwards and to the left. Then, it is easy to calculate the cost of performing a full collection in any state and acting optimally from that point forward:

$$V^F(i,l,u) = \begin{cases} L[i] + F[i,n] & \text{if } L[i] \leq S^O \\ \infty & \text{otherwise} \end{cases}$$

$L[i]$ gives the cost of the collection at i, and $F[i,n]$ gives the cost of an optimal schedule from i to the end of the trace. The condition checks to ensure that the full collection at i will fit into the old space. If not, a full collection is not valid at this point.

Calculation of the value of a young collection is more complicated, and proceeds in two steps. First, we compute $Y^y[j]$, which is the cost of an optimal schedule from i to j

using only young collections, and ending with a young collection at j. This differs from Y because we have different constraints from beginning in an arbitrary state: beginning with an old state usage of u, we must ensure that all of the data promoted from the young collections still fit into the old space. The equation becomes

$$Y^y[j] = \begin{cases} \min\limits_{i<k<j} Y[k] + y[k,j] & \text{if} u + y[l,i] + Y[k] + S^Y + \sum\limits_{i \leq h \leq j} B[h] \leq S^O \\ \infty & \text{otherwise} \end{cases}$$

The value of a young collection is then

$$V^Y(i,l,u) = \begin{cases} \min_{i \leq j < k} y[l,i] + Y^y[i,j] + L[k] + F[k,n] & \text{if } A[j,k] \leq S^Y \wedge L[k] \leq S^O \\ \infty & \text{otherwise} \end{cases}$$

The elements of the sum are

1. $y[l,i]$ – the cost of the young collection itself.

2. $Y^y[i,j]$ – the cost of a sequence of young collections until the next full collection.

3. $L[k]$ – the cost of the next full collection.

4. $F[k,n]$ – the cost of an optimal schedule from the next full collection to the end of the trace.

To be valid, the schedule must also meet a number of constraints:

1. $A[j,k] \leq S^Y$ – the allocations between the last young collection in the sequence and the next full collection must fit into the young space

2. $L[k] \leq S^O$ – the live data at the next full collection fit into the old space.

As before, if these constraints are not met for any value of j, no valid schedule contains a young collection in the current state.

Finally, the value of no collection at a certain point is reducible to the previous two cases. We simply take the minimum value of the next young or full collection in the future. That is,

$$V^N(i, l, u) = \min \begin{cases} \min\limits_{i<j} V^F(j, l, u) \\ \min\limits_{i<j} V^Y(j, l, u) \end{cases}$$

4.3.1 Complexity Analysis

We consider the asymptotic complexity of computing each array of our dynamic programming solution in turn. Since the sizes of the arrays are determined by n, the number of groups, we use n as our measure of the size of the problem. However, the actual input to the dynamic program is the table of $c[\cdot,\cdot,\cdot]$ values, which leads to the question of how the size of c relates to n. A first thought would be that c could have size $O(n^3)$ because of the possible values for the indices p, b, and d of the c table, but in fact the number of cohorts is bounded by the number of objects, which in the worst case is A/m, where m is the minimum object size (a constant determined by the language implementer, typically 4, 8, 16, 24, or 32 bytes). This will be larger than n^3 only when $G/m > n^2$, roughly. To see this, note that G/m is the maximum number of objects per group, and recall that n is the number of groups. This requires traces to be fairly short, and in any case, *asymptotically*, $O(n)$ still holds. This confirms that n is a sensible measure of the size of the problem—it determines both the size of the dynamic program's arrays *and* bounds the size of the c table given as input.

L can be computed efficiently with two copies of c, one sorted in birth order and the other in death order, so the cost to fill L is at most $O(|c| \log |c|)$. $A[i, i] = 0$ and $A[i, i + 1]$ can be determined with one pass over c. The remaining $A[i, j]$ values require a total of $O(n^2)$ work to compute. However, we really need the values only for $j - i \leq 2(S_Y/G)$, that is, if i and j are more than a certain distance apart, $A[i, j]$ necessarily exceeds S_Y. For a given i, we need entries up through the first j that gives $A[i,j] > S_Y$, but none after that. Each group adds more than $G/2$ bytes on average. In the end, we need only $O(n)$ entries of $A[\cdot,\cdot]$, a constant number per row, rather then $O(n^2)$. Given A, it is trivial ($O(n)$) to compute B.

Now consider y. For each element of c, we can add it to just those y entries for which it is relevant. Since at most $2S_Y/G$ groups (call this value g) can occur between i and j for $y[i,j]$ to be finite, $c[p,b,d]$ adds to at most $2g$ elements of y. However, g is a constant, so each element of c incurs constant work for this term. Therefore the total cost to calculate that term is $O(n)$. A similar argument works for the second term (and the two terms sum two disjoint sets of elements of c). Since each element of y takes constant time to compute, the total time to compute y is $O(n^2)$. Another way of looking at this is that $y[i,j]$ is necessarily infinite once $j - i$ is large enough. We need fill in only a constant number of diagonals.

Y is computed as the minimum of a number of cases. How many? At most $2g$ values of k can index elements of y that are not ∞, so each Y value is determined by a constant amount of work. Thus, computing Y takes $O(n^2)$ operations. This is where it is important that the second term of the sum is y, not Y. It might be more natural to instead calculate $\min_{i<k<j} Y[i,k] + Y[k,j]$. But notice that different choices of k may represent exactly the same underlying schedule of young collections, and would lead to $O(n^3)$ cost to compute Y. Since we use $\min_{i<k<j} Y[i,k] + y[k,j]$, each choice of k corresponds to a unique schedule of young collections, and we can "short circuit" the computation when $y[k,j]$ becomes infinite. The same argument applies to the calculation of $\bar{Y}$.

Each element of f requires constant time to compute given Y, L, and B, so its cost is $O(n^2)$. (One does need to precompute $\sum_{0\le j\le i} B[j]$ for each value of i in order to compute the sum term of f efficiently.) Lastly, each element of F requires $O(n)$ time to compute, but there are only n of them, so that cost is $O(n^2)$.

The total cost is dominated by a number of $O(n^2)$ terms, so is $O(n^2)$. This algorithm has undergone several rounds of revision and optimization, each time becoming more asymptotically efficient (compare it to the algorithm of (Jacek, Chiu, B. Marlin, et al. 2016), which finds only approximately optimal schedules, and has $O(n^3)$ complexity). We believe that no more room remains to increase the asymptotic efficiency of this algorithm, and therefore conjecture that its complexity is optimal for this problem, though we have not proved any lower bounds tighter than $O(n \log n)$.

The space required is $O(n^2)$.

4.3.1.1 Computing in parallel:

Given at least n processors, we can compute in parallel so as to reduce $O(n^2)$ to $O(n)$. However, there are a number of min reductions. The min computations involved for y, Y, and $\bar{Y}$ involve at most a constant number of items, so the min does not affect the asymptotic cost. In the case of f and F, however, the number of min computations can be $O(n)$ for each element, which requires $O(\log n)$ parallel time. Thus, the asymptotic cost with at least n processors is $O(n \log n)$, and the computation proceeds diagonal by diagonal. If the number of processors, p, is less than n, the cost will be $O((n^2/p) \log p)$.

What are the prospects for exploiting GPUs for this work? The large available p is helpful, but the limits on total memory or on memory bandwidth may be more problematic. We have not worked through all the details, but believe that with some cleverness about which array elements are brought into GPU memory at once, one could achieve high parallelism with GPUs. Still, GPU utilization may be reduced because of memory limitations.

4.3.1.2 Impact of varying G:

What happens if we vary the group size, G? Suppose we cut G in half. Then n would approximately double. The number of cohorts might increase some, but it is still bounded by the total number of objects. The most that the number of cohorts could change by is a factor of 8 (each of p, b, and d could split into two different values), but this can happen only until we start to approach the number of objects in the trace. The value g, our bound on the number of groups that can fit in S_Y, will double, so the cost of various computations will double. In the worst case, here is the impact on each value's computation: L, slightly more than doubled; $A[i, i + 1]$ slightly more than doubled; other A values, doubled; B, doubled; y, each element's cost is doubled (more c values are part of its sum) and twice as many elements are non-infinite, so the cost is quadrupled; Y, the cost of each element is doubled, so that the total cost will increase by a factor of 8; $\bar{Y}$ likewise goes up by a factor of 8; f, quadrupled; and F, quadrupled. Thus the overall cost has a leading term with a factor of about $1/G^3$.

4.3.1.3 Varying heap size:

If we change S_Y we need to recompute most of the values. However, S_O affects only f. Therefore we can compute y, Y, and $\bar{Y}$ once for a given S_Y, and then compute f and F for many values of S_O using the same y, Y, and $\bar{Y}$ values. f is still $O(n^2)$ to compute, but we certainly save usefully on overall computation time.

4.3.1.4 What if $G = m$?

(Recall that m is the minimum object size, so this means considering collection at each object allocation.) If we choose the smallest possible group size, then we are not grouping at all, but considering each object to be its own cohort, and not restricting where collections can occur. The cost to solve the dynamic programming problem may be very high (we typically have used $G = 2^{18}$, leading to n's being a factor of something like 5000 smaller than the number of objects in the trace). A trickier question is how different is the optimal cost for a given G from the optimal cost for $G = m$. This seems difficult to determine theoretically, though we have not worked hard on the problem. We believe there may be lower bounds we could compute that might in practice show how close the two costs must be. We also develop some empirical evidence of how the optimal cost

Table 4.1: Summary of complexity results (* = using a band matrix representation).

Variable	Cost	Impact when $G \rightarrow G/2$	When $G = m$	Parallel	Space
c	—	~ 2	—	—	$O(n)$
L	$O(c \log c)$	~ 2	$O(m \log m)$	$O(\log n)$	$O(n)$
A	$O(n)$	~ 2	$O(m)$	$O(1)$	$O(n)^*$
B	$O(n)$	~ 2	$O(m)$	$O(1)$	$O(n)$
y	$O(n^2)$	~ 4	$O(m^2)$	$O(n)$	$O(n)^*$
Y	$O(n^2)$	~ 8	$O(m^2)$	$O(n)$	$O(n^2)$
$\bar{Y}$	$O(n^2)$	~ 8	$O(m^2)$	$O(n)$	$O(n^2)$
S_Y terms	$O(n^2)$	~ 8	$O(m^2)$	$O(n)$	$O(n^2)$
f	$O(n^2)$	~ 4	$O(m^2)$	$O(n \log n)$	$O(n^2)$
F	$O(n^2)$	~ 4	$O(m^2)$	$O(n \log n)$	$O(n)$
S_O terms	$O(n^2)$	~ 4	$O(m^2)$	$O(n \log n)$	$O(n^2)$

varies with decreasing G, within the memory available to the solver. We summarize all these results in Table 4.1.

Having laid out our dynamic program in detail, we turn our attention from after-the-fact analysis to learned policies that could be used at run time.

CHAPTER 5

RUN TIME POLICIES

At each time step in the execution of a trace, we want to select the action—no collection, young collection, or full collection—that yields the lowest cost over the entire trace. Our dynamic program computes the optimal schedule: a list of which action is optimal at each step. However, this is correct only for a single program and input. A list of optimal actions for one trace does not necessarily give information about how to choose good actions in a different trace. Instead, we seek a collection *policy*—a function that takes measurable features of a program's execution as input and gives predictions of the optimal action as output. A good policy would use these features to give action predictions for previously unseen inputs to a program.

This is a difficult task. By collecting early, we can select points that have lower volumes of live data and thus lower costs. However, early collections can lead to more frequent collections, and the cost of these additional collections can eliminate any savings. Furthermore, the exact points at which collections are performed influence which objects are promoted in future young collections. These effects are complicated and may not be realized until many steps in the future. In past unpublished studies we found that *locally* optimal collection, i.e., choosing a point to collect between the current time and the time when young space has just been filled, such that young space collection cost is minimized, does *not* lead to *globally* optimal collection. One simple example of this behavior was given in Chapter 1. Globally optimal collection is a combinatorial optimization problem, and it requires exact knowledge of future behavior. However, certain features about the execution of a program, such as we use in the work reported here, can give reasonably accurate prediction of that future behavior, at least for some programs.

We tackle this task by reducing the problem of finding a policy to that of training a classifier. This approach is used in various fields, and is most often called behavioral cloning, since we wish simply to copy the optimal behavior. First, we calculate optimal schedules using our dynamic program. Then, we treat the optimal actions as class labels, and use classification algorithms to predict the optimal actions. This eliminates any need to learn policies directly from interaction with the system.

However, not all classifiers are appropriate to our task. Since they would be run at each group boundary to decide whether to collect, they must be relatively small and simple. Otherwise, the overhead of the policy itself would overwhelm any cost savings from improved collection decisions.

Even more importantly, the collection of the features imposes a greater penalty. In a practical system, the compiler would have to add code to instrument each feature used by the policy. This will add overhead for each feature, and if too many features are instrumented the additional cost will counteract any savings from sophisticated policies. To realize any cost savings, our classifiers must use a set of features that is very limited compared to the total number possible. We must include feature selection into our search for polices. On the other hand, once a feature is instrumented, keeping a buffer to record the values it has taken at past time steps is inexpensive. These historical values provide us with an additional source of information.

In the rest of this chapter, we give an overview first of the classifier algorithms we use to build policies, and then the methods of feature selection we employ.

5.1 Policy Algorithms

Here, we examine the methods we use to learn and represent policies. In all cases, if a policy selects an action that is not possible due to insufficient space in the young space or old space, we simply fall back to the default policy. Indeed, for some policies we go further, and try to learn only whether a young collection or no collection should be performed. Full collections are taken only when they are forced, as in the default policy. The idea here is that since young collections are more common, we can improve our policies by concentrating their predictive power only on the decision that will have the greatest impact.

Recall that our policy algorithms are simply general purpose classifiers that have been trained on the data produced by our dynamic program. Thus, we have attempted to choose a range of classifiers to give an overview of how different techniques may fare on our problem. We start with linear regression, which is among the simplest machine learning algorithms. Next, we employ random forests (Breiman 2001) since they are well studied in the literature and offer good performance on a wide variety of tasks. Neural networks are a very popular family of classifiers, but they are not appropriate for our needs. A large amount of computation is needed to generate predictions from neural networks, so they would introduce unacceptably high overheads if they were evaluated each time a GC system needed to make a collection decision. Of course, it is possible to use any classification algorithm as a policy, and the best choice will depend on the exact properties and constraints of a GC and the system it runs on.

5.1.1 Linear Policies

First, we present linear policies. Given a vector of features x for a certain time step in a trace, we multiply it by a weight matrix β to yield a vector of preferences p:

$$p = x\beta$$

We then choose the action for which we have the highest preference in order to implement a policy. Then, the problem of finding a policy is simply that of learning β. Broadly speaking, we take a least squares approach, and select β such that it closely approximates a target value y. The targets we use take two different forms. The simplest is to use the state-action values for all valid actions along the states visited in an optimal schedule:

$$y = (V^N, V^Y, V^F)^T$$

This is the approach taken by the well-known LSTDQ algorithm from the reinforcement learning literature, though it learns from traces of interaction with the system rather than directly from known state-action values.

Note that this strategy has an important drawback. The state-action values are generally all larger toward the beginning of a trace than at the end, since more actions remain to be taken. However, since the policy chooses the action with the largest preference, we only need the *relative* rather than absolute preference values. All together, this means that a large portion of the expressive power of the policy could be wasted on essentially predicting how far a certain state is from the end of the trace: a useless fact. Instead, we can use relative values themselves as the training target:

$$\begin{aligned} m &= \min(V^N, V^Y, V^F) \\ y &= (V^N - m, V^Y - m, V^F - m) \end{aligned}$$

The exact method we use to calculate β given these targets is intertwined with the feature selection algorithm we use in the linear case, so we defer discussion of it to Section 5.2.1.

5.1.2 Hellinger Tree Policies

Our second type of learning algorithm is based on decision trees. Each internal node of the tree holds a feature ID number and a threshold. To decide which action to take at a time step, we begin at the root node, and compare the current value of the feature against the threshold. If it is less than or equal, we proceed to the left child of the node, and if not, we proceed to the right child. Eventually we reach a leaf, which lists the probability that each action should be taken.

Decision trees are built recursively by greedily selecting the feature and threshold that best splits the examples to be classified into left and right groups according to some measure. The most common measures, Gini impurity and information gain, are not suitable for our purposes. This is because the distribution of the actions in our data are highly skewed. In optimal schedules, most of the time no collection should be performed. Young collections are rare, and full collections rarer still.

To overcome this limitation, we rank splits according to the Hellinger distance, which is much more robust to skewed distributions (Cieslak et al. 2012). To see how it is calculated, consider the case where we are distinguishing between only two classes; denote them $-$ and $+$. A given split creates two conditional probability distributions. Each represents the chance that an example will be passed along to either the left or right child, given that it belongs to a given class. The PMF (probability mass function) of one distribution is given by $P(L|-)$ and $P(R|-)$, and the PMF of the other is given by $P(L|+)$ and $P(R|+)$. The Hellinger distance is then one measure of the distance between these two distributions. The calculation is:[1]

$$d_H = \sqrt{1 - \sqrt{P(L|-)P(L|+)} - \sqrt{P(R|-)P(R|+)}}. \tag{5.1}$$

To visualize the Hellinger distance, note that these two distributions form vectors in a two-dimensional space. The Hellinger distance is then the Euclidean distance between the square roots of these two vectors. It reaches its maximum value, 1, when the two distributions are completely disjoint, and is zero if the distributions are equal. We select the split with the largest distance between the distributions it induces. Note, however, that

[1]This formula differs by a constant factor from what is often seen in the literature. Because we are interested only in finding the point that maximizes the Hellinger distance, the constant factor is irrelevant for our purposes.

we have three actions, and therefore three classes. This gives us three different conditional probability distributions. There is little guidance in the literature as to the best way to combine the distances between these three points into a single measure of the quality of a split. We take what is arguably the simplest option, and use the sum of the three pairwise distances between the three distributions.

As we build the tree we split the examples into smaller and smaller groups. Eventually, either a given group contains only one action, or no feature and threshold will further subdivide it. In this case, we add a leaf that records the distribution of actions in the group. We use Laplace smoothing, a technique that helps to avoid overfitting. It adds one to the count of examples in each class. This reduces the confidence the classifier has in the empirical probabilities when the number of examples at a leaf is small, and ensures that at every leaf every action has at least a small probability.

Now that we have presented the policy algorithms we use, we discuss our various methods of selecting features to feed into them.

5.2 Feature Selection Algorithms

In this section, we explain the feature selection algorithms that we use. As discussed in the preceding literature review (Guyon and Elisseeff 2003), feature selection algorithms can be divided into three broad families. In order to survey the possible methods that can be applied to our task, we use one method from each family. The first is embedded methods. These are built-in to larger algorithms that both choose features and train classifiers as part of the same process. Group Orthogonal Matching Pursuit, our first feature selection algorithm, falls into this category. Next are wrapper methods, which repeatedly train an underlying classifier as a subroutine using different features. Those that yield the best performance of the classifier are selected. From this group, we use Random Forest Importance. Finally, information theoretic feature selection algorithms directly compare the features themselves against the class to be predicted in order to calculate which features may be most informative. Our final feature selection algorithm, the Mutual Information Filter, is of this type.

5.2.1 Group Orthogonal Matching Pursuit

This first feature selection algorithm is specific to linear policies, as in Section 5.1.1. Recall that we seek a matrix β that, when multiplied by a vector of features x corresponding to a particular state, yields

$$p = x\beta$$

where p is a vector of preferences for each action. The policy then selects the action with the highest preference. Group Orthogonal Matching Pursuit (Swirszcz, Abe, and Lozano 2009) is an algorithm that provides an elegant way of both selecting a small subset of features and calculating β. It proceeds as follows.

- Take P and $\mathcal{X}$ as input. P denotes the ground-truth matrix, which is ether the matrix of state-action values or relative values, depending on the policy algorithm we are using. $\mathcal{X}$ is a set of feature matrices X_i where each row of the matrix represents the current and historical values the ith feature takes at a particular time point.
- Initialize R, the residual, to be P, and $\mathcal{S}$,the set of selected features, to be empty.
- Repeat until the desired number of features have been selected.
  - Calculate the length of the projection of R onto the column space of each feature matrix, and select the feature that maximizes this value. In other words, choose $X_i \in \mathcal{S}$ such that $X_i^T X_i^\dagger R$ is maximized.[2]
  - Add feature i to the set of selected features – $\mathcal{S} \leftarrow \mathcal{S} \cup \{i\}$
  - Assemble $X_\mathcal{S}$ by concatenating the matrices of the features that have been selected so far.

[2]$\dagger$ denotes the Moore-Penrose pseudoinverse.

  - Calculate the current least-squares estimates of P and β – $\beta \leftarrow X_{\mathcal{S}}^{\dagger} P$ and $\hat{P} \leftarrow X_{\mathcal{S}}\beta$.
  - Update the residual $R \leftarrow P - \hat{P}$. This prevents redundancy among the selected features by ensuring that the subsequently selected features can represent the component of P that is orthogonal to the space spanned by the features that have already been selected.

- Return $\mathcal{S}$.

5.2.2 Random Forest Importance

Importance is a concept originally developed to investigate the internal workings of random forests (Breiman 2001), but that has also been used as a method of feature selection (Archer and Kimes 2008). For each tree of a random forest, a training set is chosen by randomly selecting examples from the full training data with replacement. The examples that are not selected then naturally form a different random test set for each tree. We can use these sets to calculate the importance of each feature.

First, we calculate the classification accuracy of the tree on the test examples. Then, we randomly permute the values that a certain feature takes among these examples, and we calculate the tree's classification accuracy again. When these two are compared, the more that permuting the values decreases the classification accuracy, the more important we conclude the feature is to the classifier. The overall importance of a feature is then the average of its importance to each tree in the random forest. We then select the most important features for use later in our final classification algorithm.

Note that it is typical to build each tree in a random forest using a random subset of features, as well as a random subset of training examples. When we attempted to do this on our data, most trees did no better than random at classifying training examples. So, we were unable to calculate meaningful importance values. This is likely because, as discussed in Chapter 6, the vast majority of the features do not carry information relevant to our task. Instead, as we calculate feature importance, each node in a tree is able to select greedily a feature and threshold for its split from the entire set of features and values in the data.

5.2.3 Mutual Information Filter

Our last feature selection technique is adapted from Fleuret (2004). It is based on mutual information, which is one measure of the similarity between random distributions. Mutual information takes its maximum value when one distribution is a deterministic function of the other; one variable can be used to calculate the other, and the distributions contain the same information. Conversely, if the two distributions are independent, the mutual information between them is zero. Furthermore, conditional mutual information tells us how much more information one distribution gives about another, given that a third distribution is already known. This motivates the algorithm. We wish to select features that have a large mutual information with, and are thus informative about, our target distribution. However, using the conditional mutual information given the previously selected variables protects us against choosing features that are highly redundant.

If we let $H(X)$ represent the entropy of the distribution X, then the mutual information between X and Y is $I(X, Y) = H(Y) + H(X) - H(X, Y)$, and the mutual information between X and Y conditioned on Z is $I(X, Y|Z) = H(X, Z) - H(Z) - H(X, Y, Z) + H(Y, Z)$. These values can be easily calculated by counting the occurrences of values that the distributions jointly take. However, this can only be done with discrete distributions. Our ground truth distribution, which is the schedule of optimal actions, is discrete, but our feature values are not. They are comprised of counts that may be very large, and are therefore effectively real-valued. To overcome this, we discretize each feature's values into ten equally sized bins that range from zero to the maximum value the feature takes in the training data. The algorithm to select features is then as follows.

- Take P, the schedule of optimal actions, and $\mathcal{X}$, the set of discretized feature vectors as input.
- Find the feature that has the highest mutual information with P. That is, calculate $X_j = \arg\max_{X_i \in \mathcal{X}} I(P, X_i)$.
- Initialize the set of selected features: $\mathcal{S} \leftarrow \{X_j\}$. We begin by selecting the feature that is most informative about the optimal schedule. Note that for the mutual information calculations, we treat the present and historical values of a feature as separate distributions. So, here we are adding one feature, but possibly more than one distribution, to the set.
- Repeat the following until the desired number of features is selected.
  - Find the feature with the highest mutual information with P conditioned pairwise on each feature already chosen:

    $$X_k = \arg\max_{X_i \in \mathcal{X}} \min_{X_j \in \mathcal{S}} I(P, X_i | X_j)$$

    The outer argmax selects the most informative feature, but the inner minimization ensures that it is not highly redundant with any previously selected feature. This is done only pairwise since conditioning on more than one feature would be computationally intractable.
  - Add the selected feature to the set: $\mathcal{S} \leftarrow \mathcal{S} \cup \{X_k\}$.
- Return $\mathcal{S}$.

Finally, we turn our attention to empirical results showing the performance of our algorithms.

CHAPTER 6

EMPIRICAL RESULTS

We now present empirical results of running our various algorithms. First, Section 6.1 gives the output of our dynamic program, establishing a lower bound to the performance of any policy. Then, we give empirical results for our linear and decision-tree based policies in the following sections.

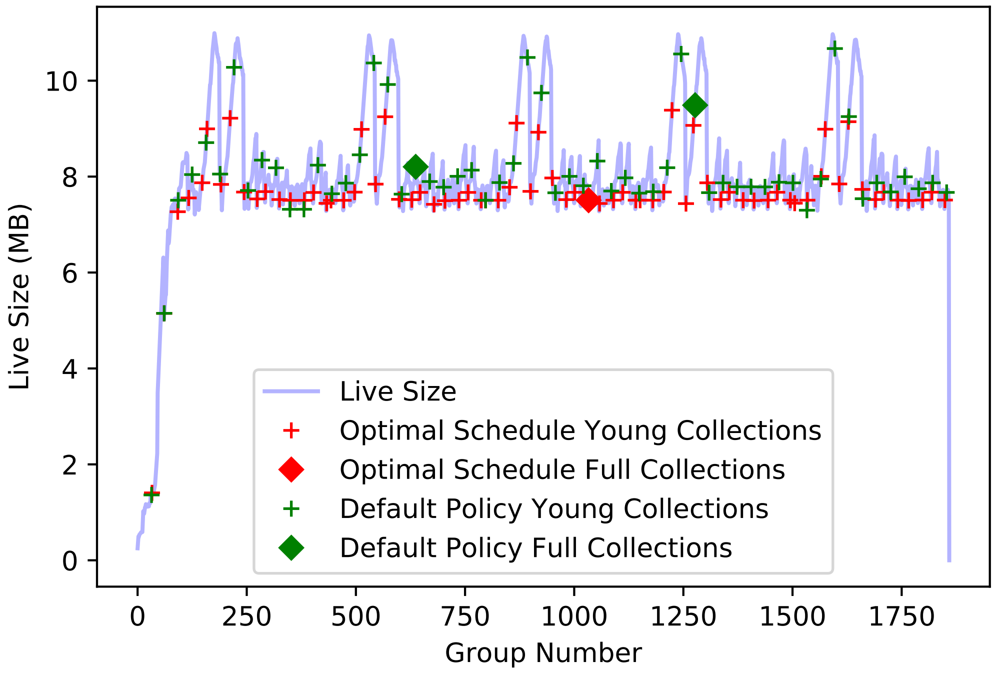
6.1 Dynamic Program Results

6.1.1 Heap Size and Collection Cost

To begin, we consider results for a particular trace: a run of `javac` on one input, using a young size of 8 MB. We chose this trace because its relatively short length makes various properties of its optimal policies easily visible.[1] First, Figure 6.1 shows the points where collections occur in the optimal schedule and under the default policy, using an old size of 25 MB. Notice that the young collections in the optimal schedule tend to occur at smaller live sizes than those in the default policy, but they are not always located at exact local minima of the live size curve. This illustrates that minimizing the total cost of a schedule depends on global, rather than simply local, information about the trace. However, also note that the optimal schedule contains only one full collection. This contrasts with the two expensive full collections that the default policy performs, greatly increasing its cost.

[1]We make no claim as to whether the heap sizes are what one would choose in practice. This example is chosen only to illustrate how optimal collection can be different from the default policy.

Figure 6.1: Collection points for a particular trace.



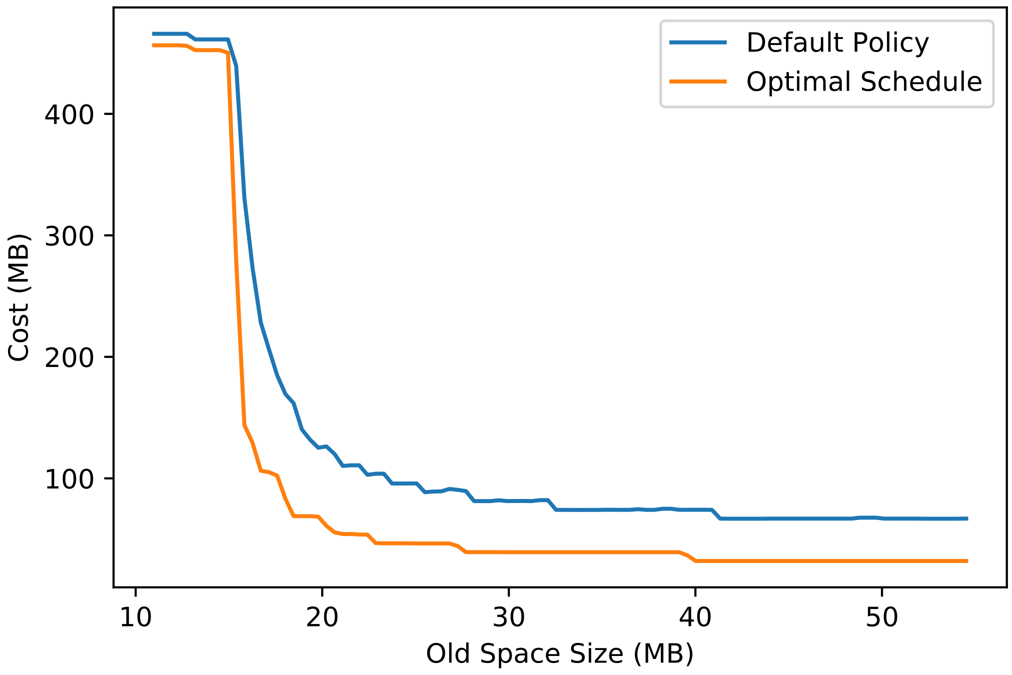
Next, Figure 6.2a plots the cost of optimal and default schedules against the old space size. As expected, the optimal cost is always less than the default cost, and in general costs decrease as the heap size grows larger. However, note that the default cost can sometimes be *greater* at larger heap sizes, while the optimal cost never increases. This is because at larger heap sizes, at worst, the optimal schedule will simply collect at the same points as it will for smaller heap sizes. It may seem counter-intuitive that increasing heap size can cause the default policy to make decisions that cost more. However, this is a (now) well-known effect and comes about because collection points shift and can cause a full collection at a time with high live size shortly before a large number of objects becomes unreachable. Adjusting young collection times can avoid that costly full collection.

As for the optimal schedules, increasing the heap size never excludes schedules that are valid at smaller sizes. Also, note that full collections tend to be much more expensive than young collections. As the heap grows, it provides opportunity for more inexpensive placement of collections. But when certain thresholds are reached, the trace can complete with one fewer full collection, giving a substantial decrease in costs. This explains the characteristic stair-step shape of the cost curves. The cost decreases slowly until an entire full collection is optimized away, at which point the cost drops quickly. Note, furthermore, that these points occur at smaller old sizes for the optimal schedule than for the default schedule. Finally, notice that the optimal schedule curve extends further left than the default schedule curve. This shows that careful placement of collection points can allow a collector to continue in situations where the default strategy paints itself into a corner with no valid actions left to take. We present the same data again in Figure 6.2b, scaled by the default schedule cost (at the same heap size). As the old size changes, the relative savings of the optimal schedule varies within certain bounds.

The previous graphs tell us the cost of the two different schedules at various old sizes, but we can ask a different, related question: If one is willing to tolerate a certain cost of collection, what is the smallest old size one can use without exceeding this cost? Fig-

Figure 6.2: Cost vs old size for a particular trace

(a) Cost has not been scaled



(b) Cost scaled to fraction of default cost

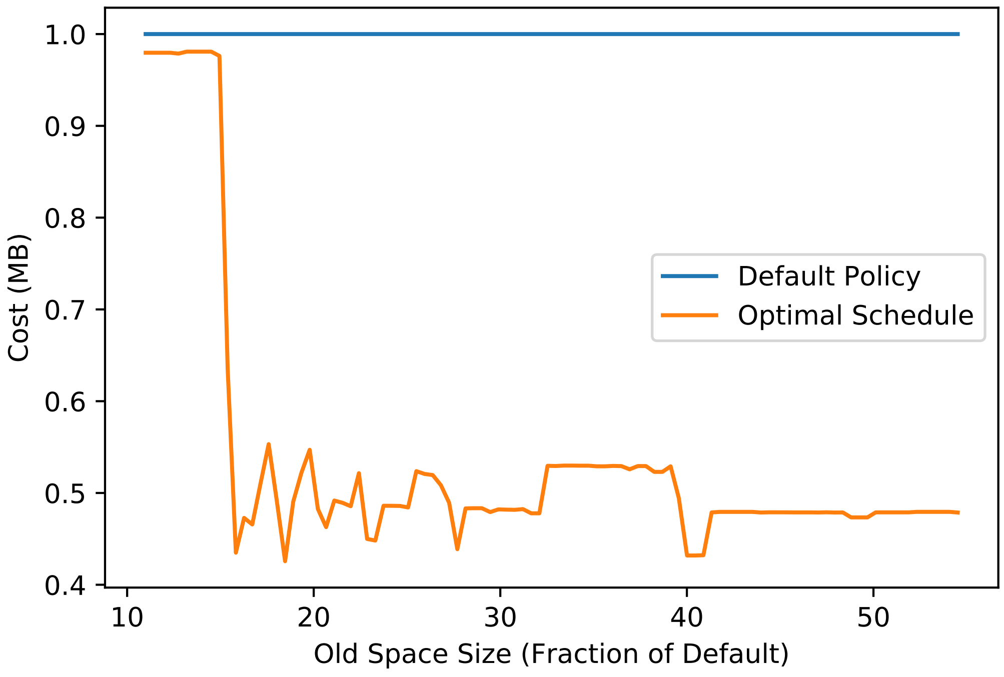
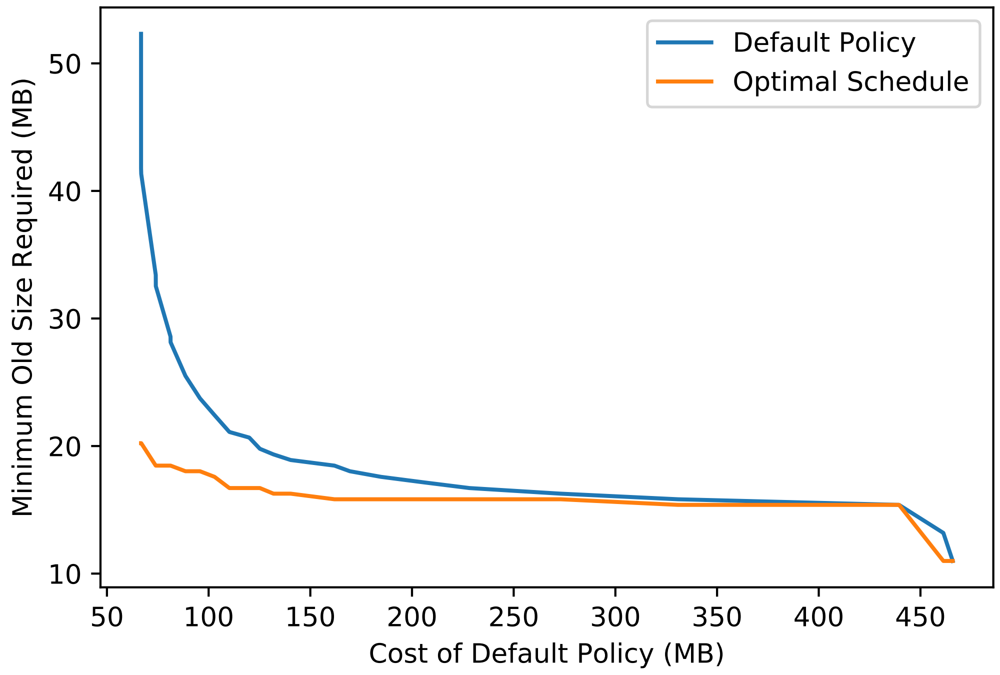
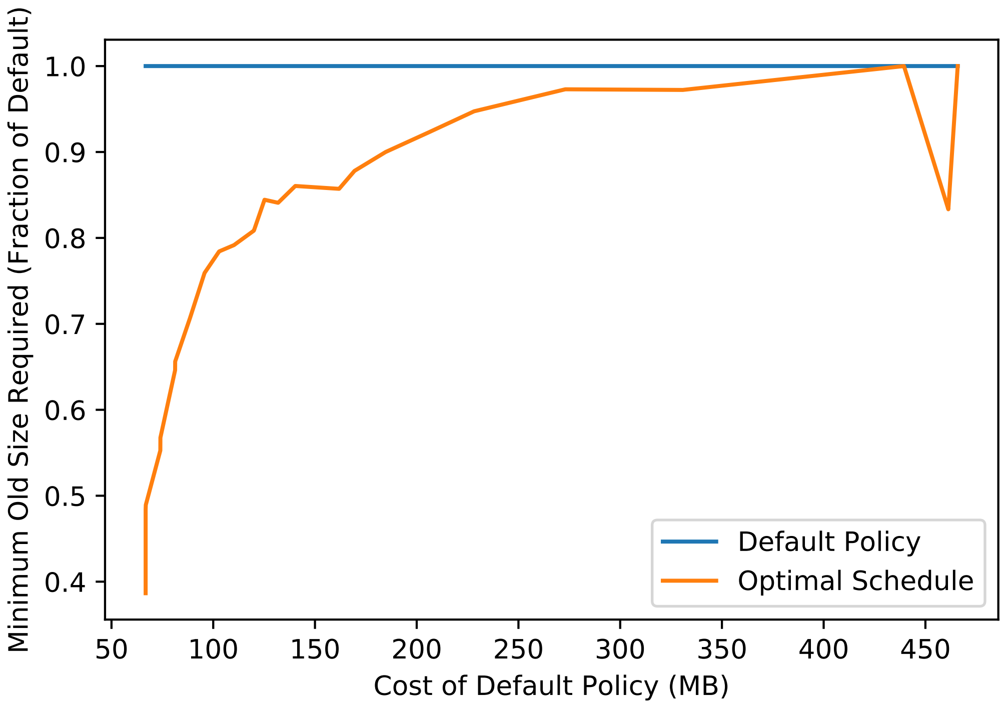


Figure 6.3: Minimum required old size vs cost for a particular trace

(a) Size has not been scaled



(b) Size scaled to fraction of minimum required for default policy



ure 6.3a answers this question. Using the same trace and settings as the previous figures, the horizontal axis shows the cost we are willing to tolerate, and the vertical axis shows the minimum old size needed to meet this requirement. As the cost decreases, the old size needed increases, but it increases much more quickly in the case of the default policy. Figure 6.3b presents the same data, scaled by the size required by the default schedule. Note that the curves meet in the upper right corner. This shows that if we tolerate a very high cost, we run into the hard limit that the live data must fit into the heap, and we cannot reduce the heap size any more. Essentially, we must collect at nearly every time step, and no optimization is possible.

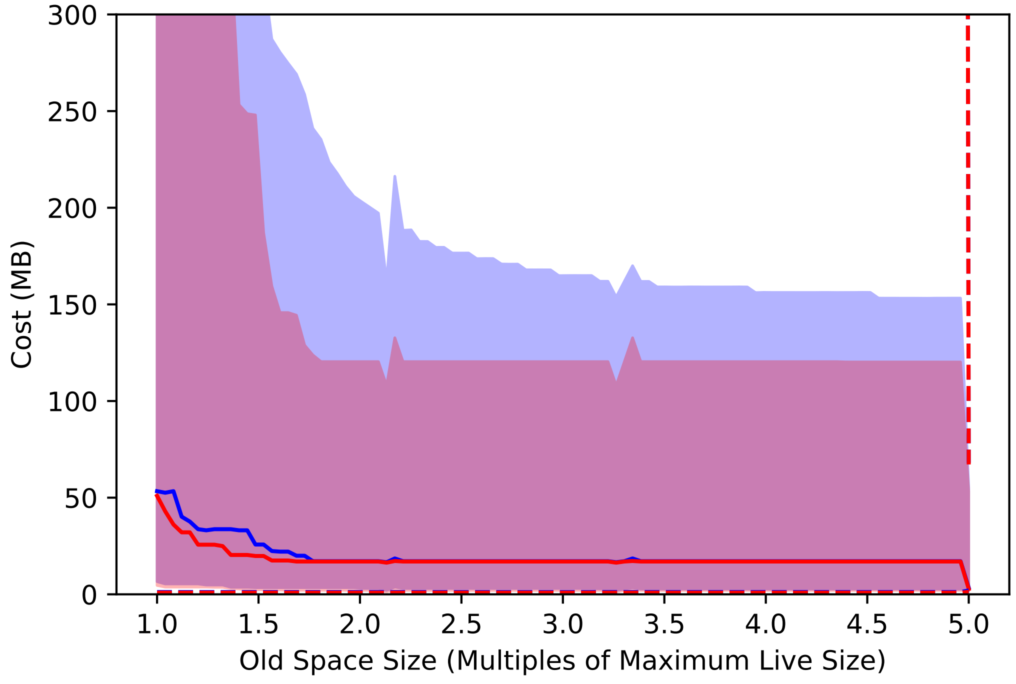
Next, we expand our view from a single trace to the cost reductions associated with optimal schedules on a wide range of programs and inputs. Figure 6.4a shows the relationship between the costs of the default and optimal schedules versus the size of the old space. These results are summaries over all of the programs, inputs, and young sizes we analyzed; in order to draw meaningful comparisons across different traces, the old sizes on the horizontal axis are scaled to be multiples of the maximum volume of live data in the trace ("max live size"). Clearly, the optimal schedule provides a greater cost savings at smaller old sizes, but there is quite a large variance in the data.

Figure 6.4b shows the same data as Figure 6.4a, but the cost of each optimal schedule has been divided by the cost of the default schedule that corresponds to the same trace and settings. The savings become more clearly visible. At larger heap sizes, the average optimal schedule costs about eighty percent of what the default schedule does, and the relative cost decreases at smaller old sizes. This matches our intuition that smaller heap sizes are "tighter": they are more sensitive to the placement of collections, and thus provide more opportunity for optimization.

As we did for the single trace, we can ask what minimum old size is needed for a given cost; these results are summarized in Figure 6.5. In order to compare between traces of different lengths, the horizontal axis is scaled to show the cost per byte allocated, some-

Figure 6.4: Summary of cost vs old size

(a) Overall cost vs old size



(b) Overall Cost vs old size, scaled to multiples of the cost of the default policy

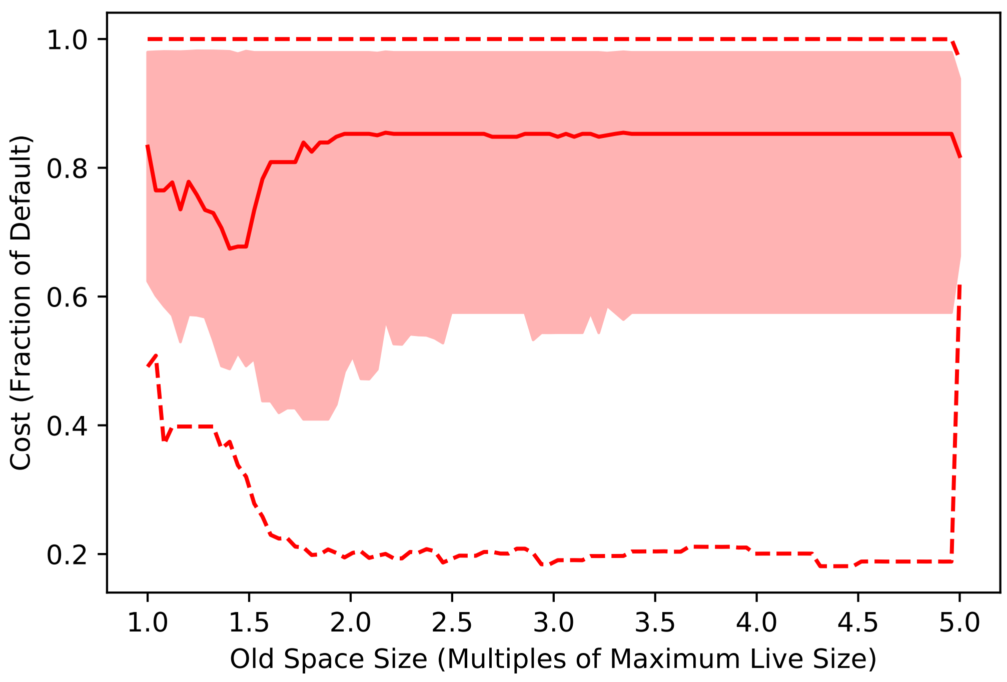
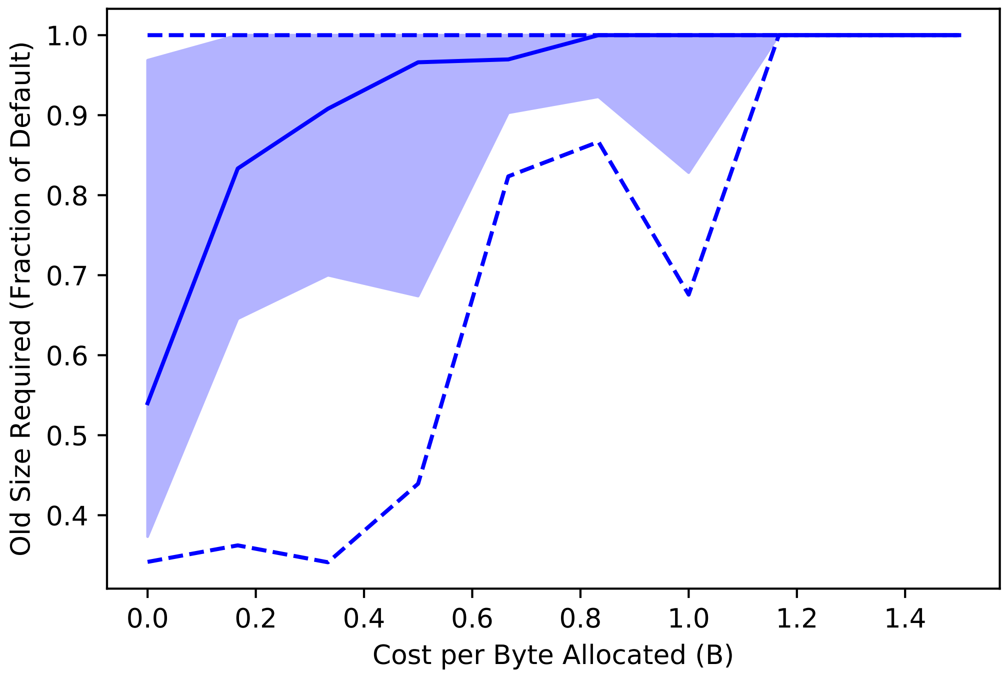


Figure 6.5: Overall min old size vs cost, scaled to fractions of the default size.



times called the mark/cons ratio. Another, possibly more intuitive, way to describe that quantity is the average number of times an allocated byte will be copied by the collector.

6.1.2 Impact of Collector Model

The model of a generational collector we presented in Chapter 3 differs slightly from that which we used in Jacek, Chiu, B. M. Marlin, et al. (2019). There, we used what we will call an "unsafe" model, where a young collection is allowed if the objects it promotes will fit into the old space. Of course, how much of the data in the young space will be promoted cannot be determined beforehand. A collector running on this model would need a mechanism for backing out of a failed collection and performing a full collection instead. We call the alternative model we use here "safe" because any young collection that is allowed to begin will have enough room in the old space to complete successfully, which we feel is more realistic for a practical collector. Empirical results comparing the optimal schedules under these two collector models are presented here.

First, Figure 6.6 compares the costs of the default policy against the optimal schedules under these two different models. At very small heap sizes, the additional flexibility given by the unsafe model allows for much less expensive schedules. But at moderate and larger sizes, the optimal schedule costs for the two different models are nearly identical.

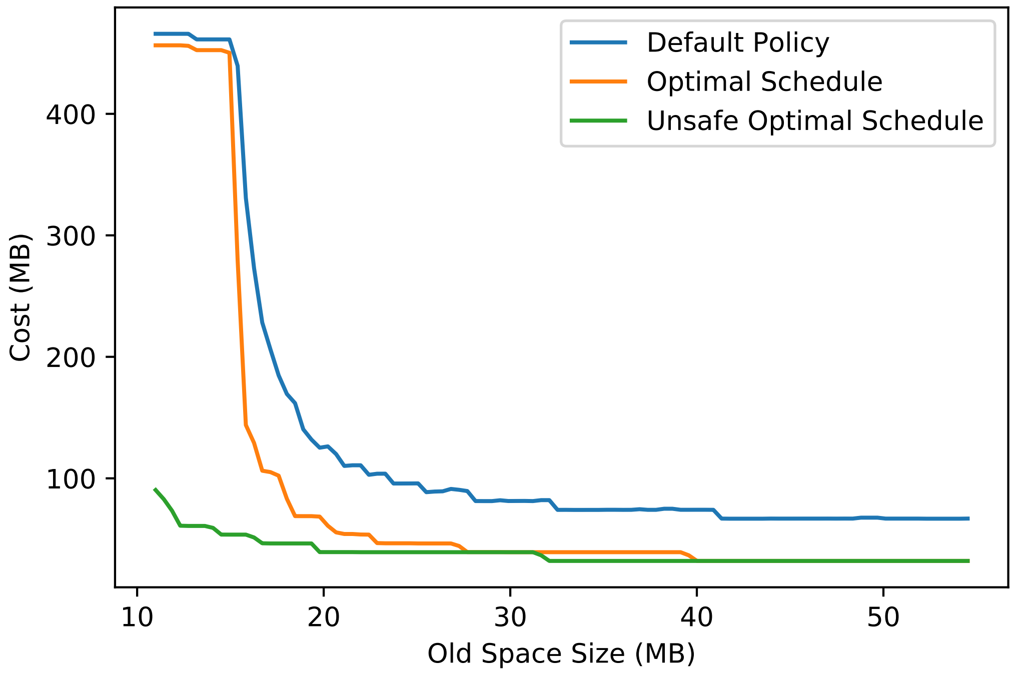
This trend is common among all of our traces, as seen in Figure 6.7. A large advantage for the unsafe model at very small heap sizes mostly disappears at medium and large sizes. We conclude that, for our purposes, the choice between these two collector models is largely irrelevant.

6.2 Learned Policies

As we have established theoretical lower limits to the costs incurred by generational garbage collectors, we now turn our attention to the performance of learned policies.

Figure 6.6: Old Size vs Cost Using Both Collector Models for A Particular Program

(a) Cost has not been scaled



(b) Cost scaled to fraction of the default cost

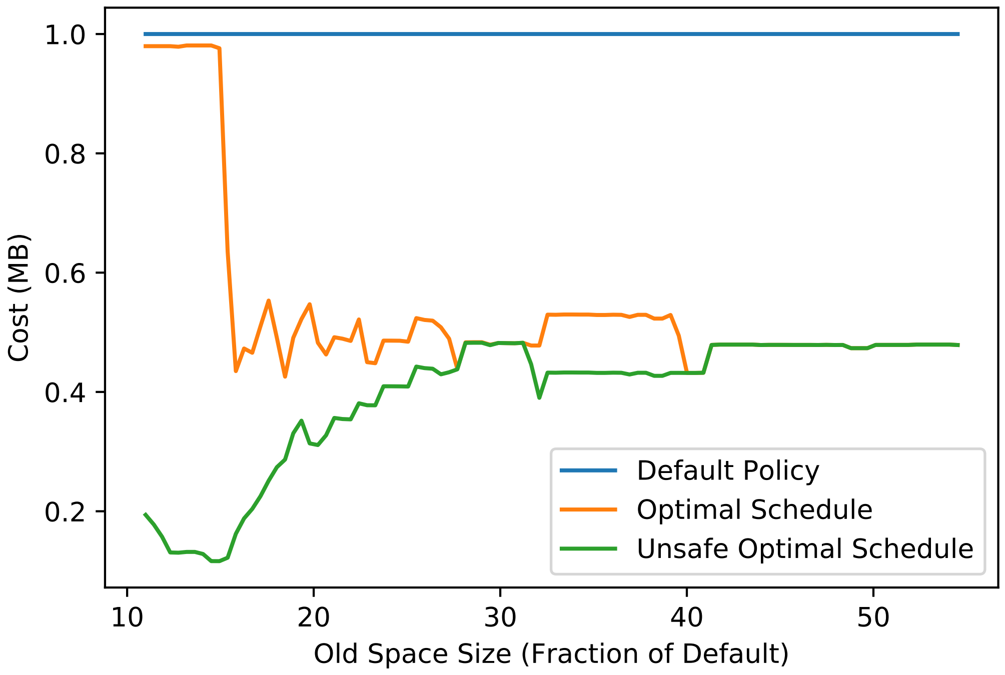
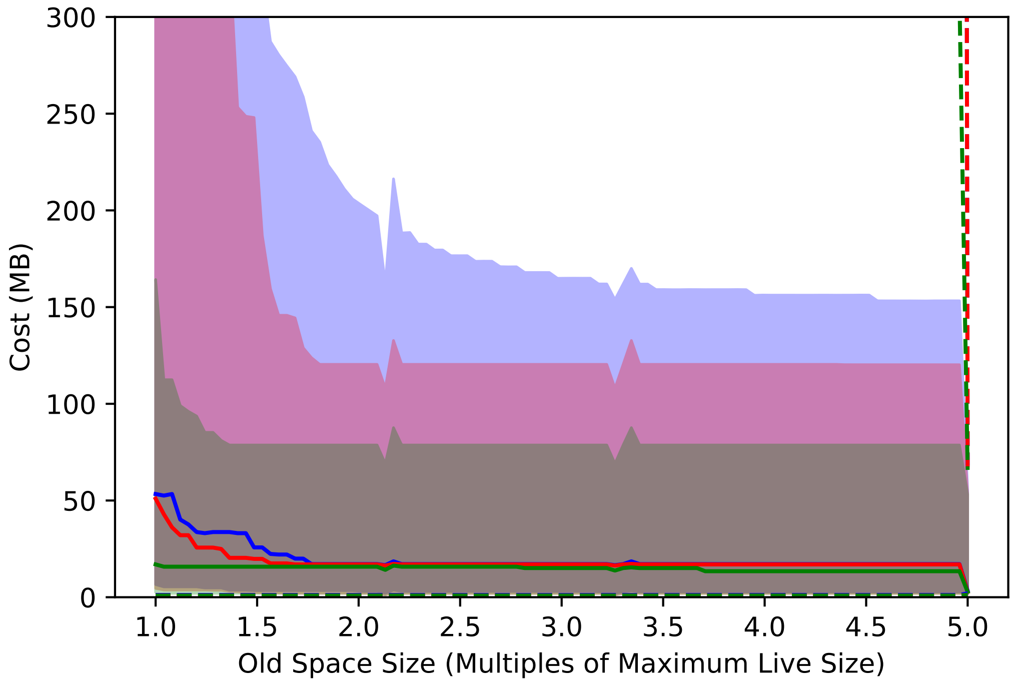
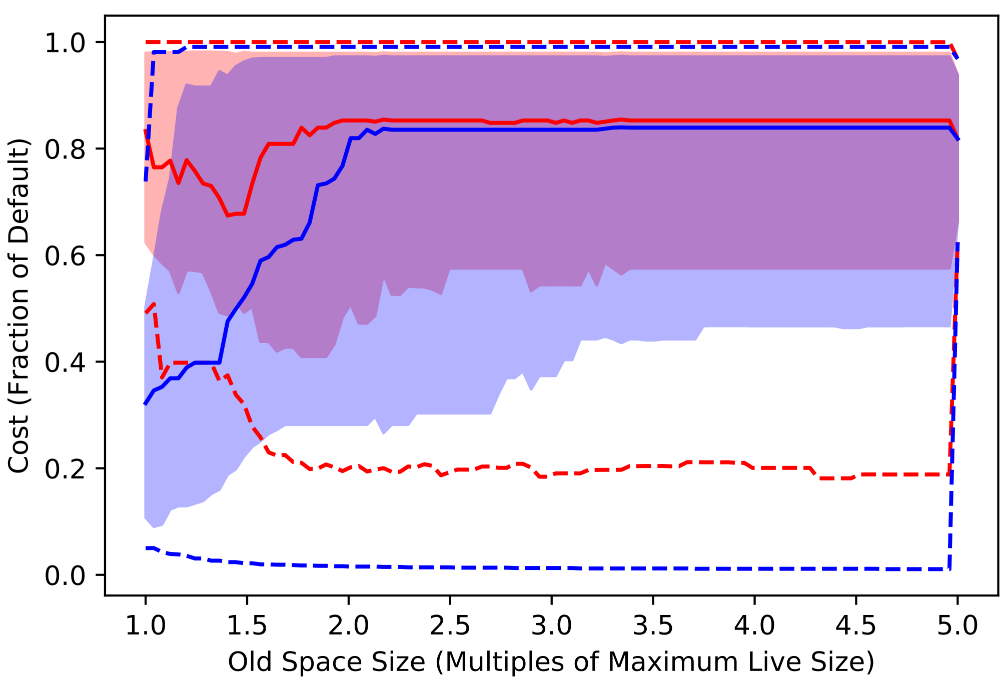


Figure 6.7: Summary of Old Size vs Cost Using Both Collector Models

(a) Old size has not been scaled.



(b) Old size scaled to fraction of the minimum old size required by the default policy



For policy training, we selected 32 instrumented features as an approximation of the number that could be used in a practical system. The fraction of the young and old spaces that are occupied would also be available to a system so they are also included, bringing us to a total of 34 features. First, we first trained a single decision tree on each trace in our set using features selected based on importance, and then tested its performance on that same trace. In all cases, the learned trees were able to perfectly reproduce the optimal policy. This likely indicates that they were overfitting to their training data, but it does suggest that the selected features are informative enough to represent good policies.

Next, we investigated how well our learned policies could generalize across different inputs to a program. For the cases that we used random forests, each was built with 100 trees. We selected this large number in order to give our policies the best chance of performing well, since random forests do not tend to overfit as the number of trees they contain is increased. On the other hand, each additional tree in a policy increases the cost of making each action decision. A practical system would require additional engineering to precisely quantify these costs and make any necessary trade offs between the cost and performance of a decision tree policy. Still, our results give insight into what might be possible for any practical system.

In all results reported below, we performed leave-one-out cross validation of our policies by training on all but one input to a program. The resulting policy was then simulated on the remaining input. We repeated this procedure for every input in our suite.

Our training data represented optimal schedules for old spaces sized to be three times the maximum live size of the trace. For all programs except `luindex`, we used a young space size of 8MB. This size is large enough that `luindex` did not perform any collections, so instead we used a young space size of 2MB for that program only.

6.2.1 Generalization Over Old Space Sizes

If a policy is effective at one heap size, will it perform similarly well at other heap sizes? This is the question we answer in this section. To do so, we construct a specialized policy for each program and input from the optimal schedule calculated for a heap size of three times the trace's maximum live size. When run at different old space sizes, the policy will attempt to follow the same schedule, taking the same action at each time step. If that action is not possible, the policy falls back to the action the default policy would take. We call this a "schedule-policy" because it is a policy that simply tries to follow a pre-established schedule.

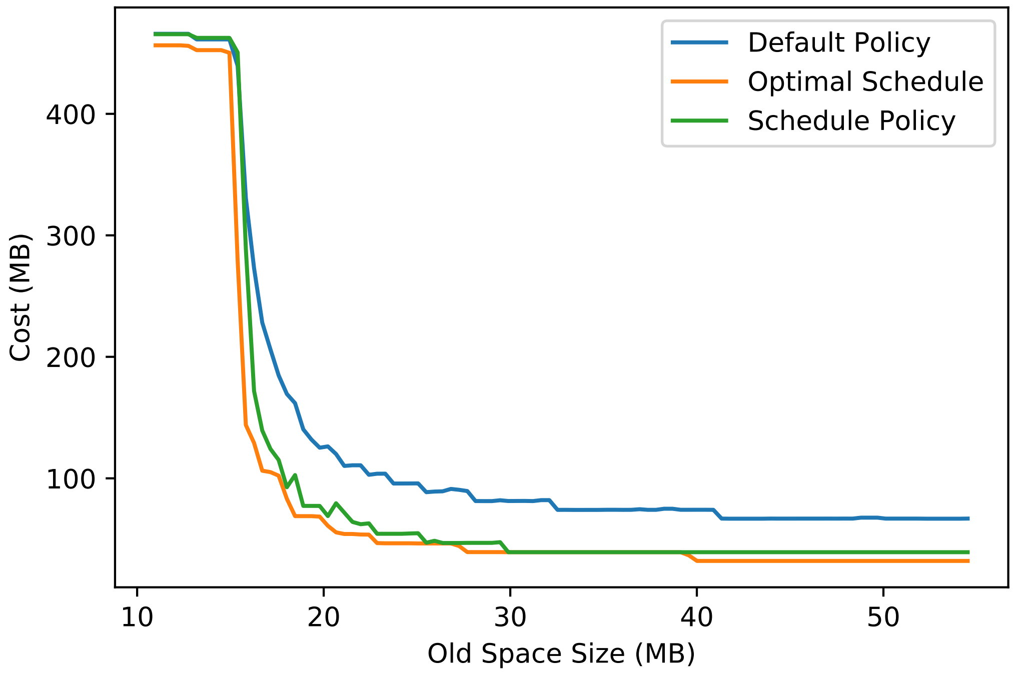
Of course, this is not a practical policy; one rarely needs to run the same program on the same input, just using different heap sizes. Instead, the idea here is to establish how easily policies can generalize over different old space sizes. For example, do policies need to be narrowly tailored to specific heap sizes, or is a policy trained at one heap size likely to overfit and perform poorly at other sizes?

Consider Figure 6.8, which compares the cost of this policy to the default and optimal on the same `javac` program and input considered previously in this chapter. As we can see, the schedule policy closely follows the cost of the optimal schedule.

The performance for this policy is summarized across all programs and inputs in Figure 6.9. At very small heap sizes, considerably smaller than the three times the maximum live size they are trained at, the average cost of the schedule policy degrades, approaching the default. Additionally, in a small number of cases, this policy can cost considerably more than the default. However, at most old space sizes, this policy essentially matches the performance of the optimal schedule. In other words, to build a nearly optimal policy for a particular program and input at any heap size, all one needs to do is memorize an optimal schedule for one particular old space size. We can expect policies to easily generalize over different old space sizes.

Figure 6.8: Schedule-Policy Performance for a Particular Trace

(a) Cost has not been scaled.



(b) Cost scaled to fraction of default policy cost.

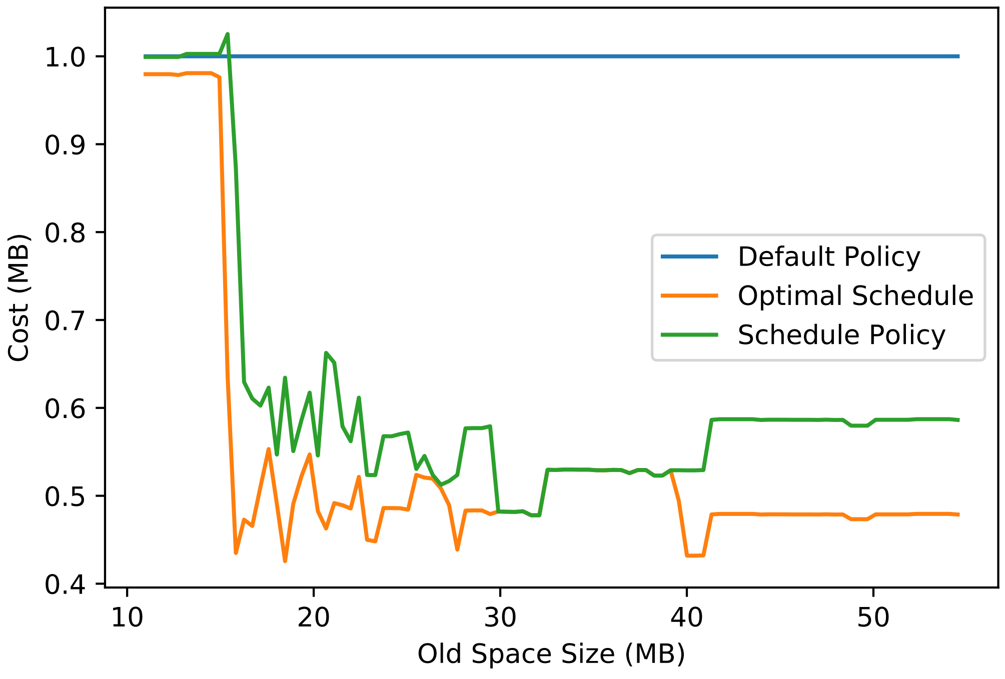


Figure 6.9: Schedule-Policy Performance Summary

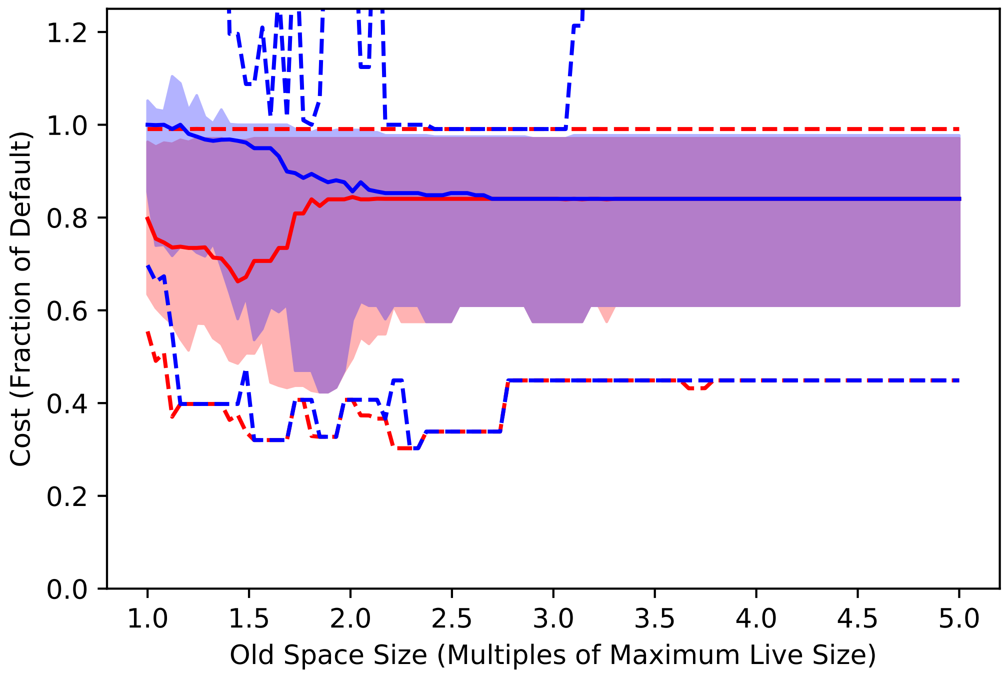
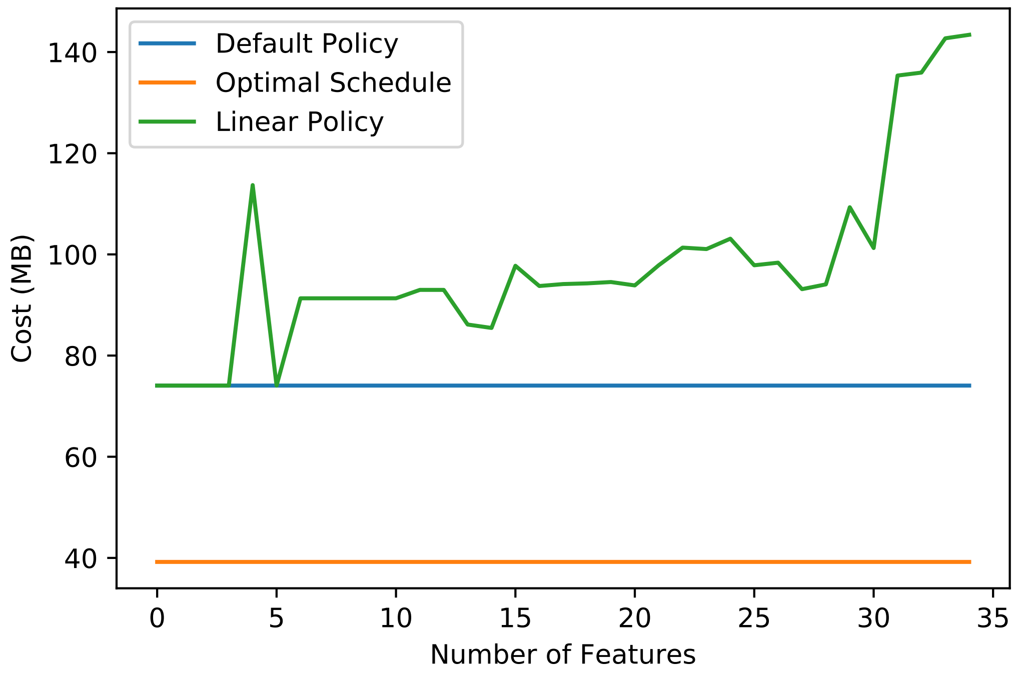


Figure 6.10: State-Action Value Linear Policy on a Particular Trace



6.2.2 Linear Policies

Here, we presents results from the performance of linear policies on our traces. Recall that we have two different linear policies. Both use the group orthogonal matching pursuit algorithm, but one is trained using the state-action values as their target, and the other uses the disadvantage. Figures 6.10 and 6.11 show the performance of these policies on the `javac` trace we have been using for illustration, and Figures 6.12 and 6.13 summarize the performance of these policies over all traces. Finally, Figures 6.14, 6.15, 6.16, and 6.17 recapitulate the same results for policies that use history features reaching twenty time steps into the past.

In all cases, the graphs show a consistent pattern. The policies are able to reproduce the performance of the default policy when using up to three features, but at greater numbers, their cost rapidly increases. Only in rare instances are they able to perform at lower cost than the default. We conclude that linear polices are ineffective for this problem. When

Figure 6.11: Disadvantage Linear Policy on a Particular Trace

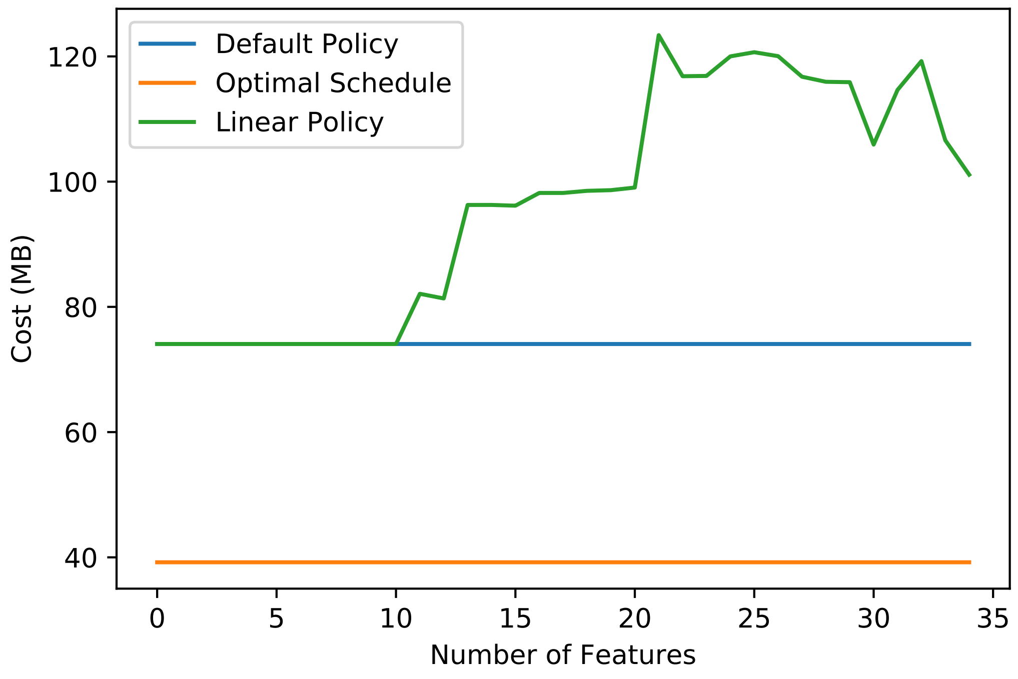


Figure 6.12: State-Action Value Linear Policy Summary

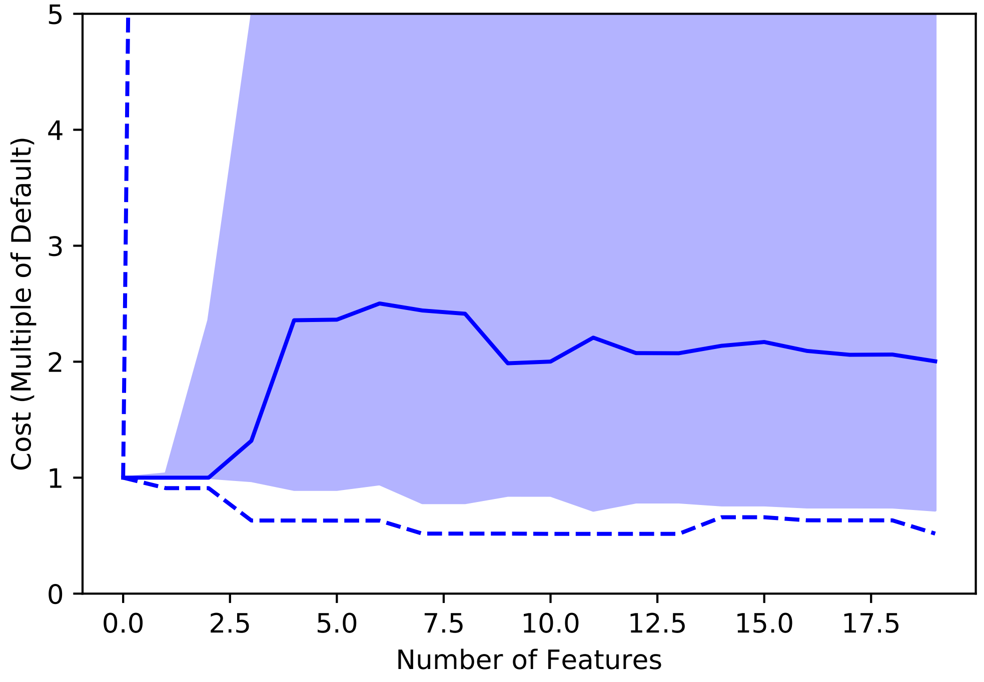


Figure 6.13: Disadvantage Linear Policy Summary

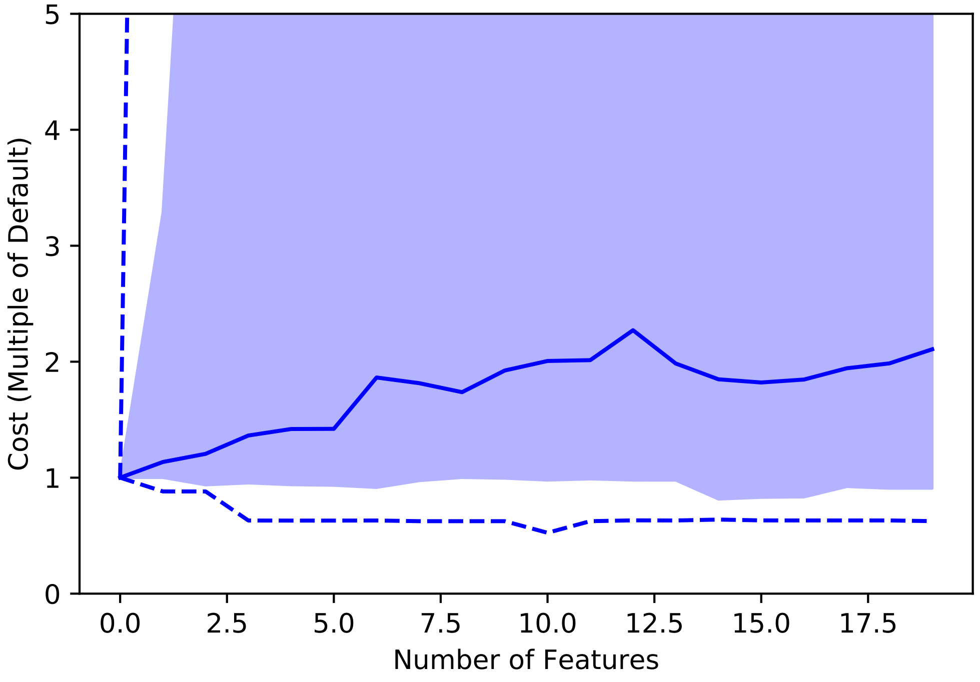


Figure 6.14: State-Action Linear Policy with History For a Particular Trace

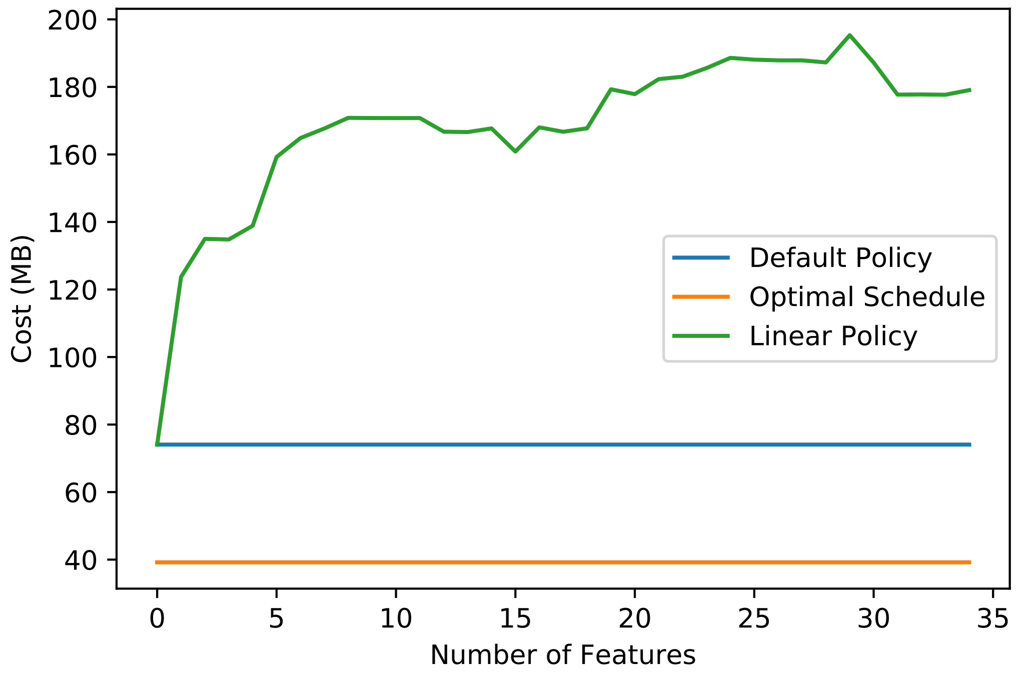


Figure 6.15: Disadvantage Linear Policy with History for a Particular Trace

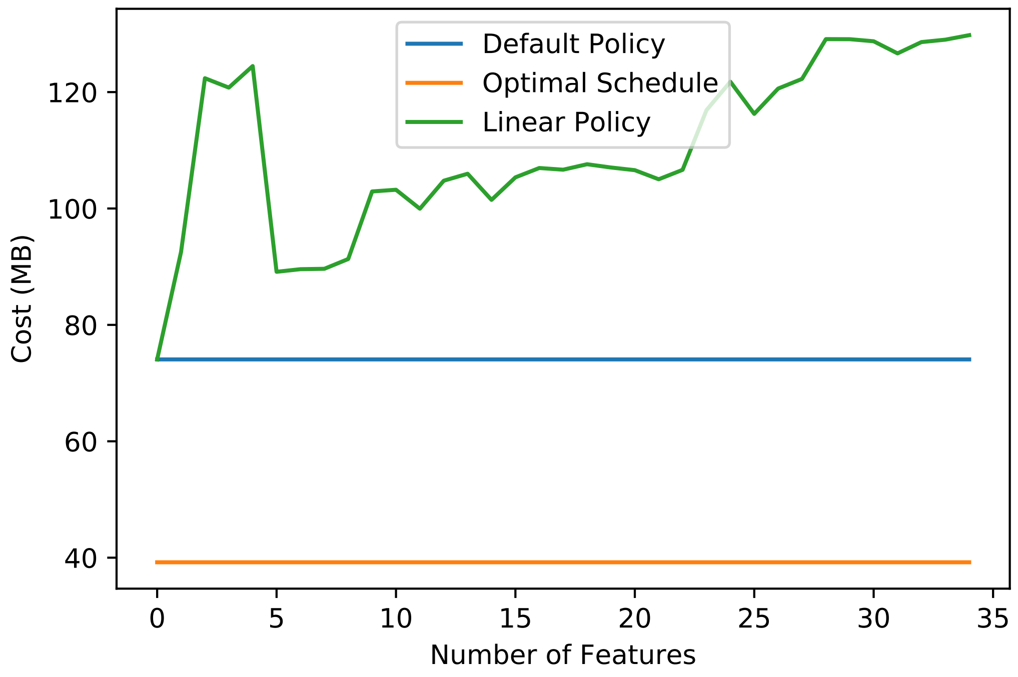


Figure 6.16: State-Action Linear Policy with History Summary

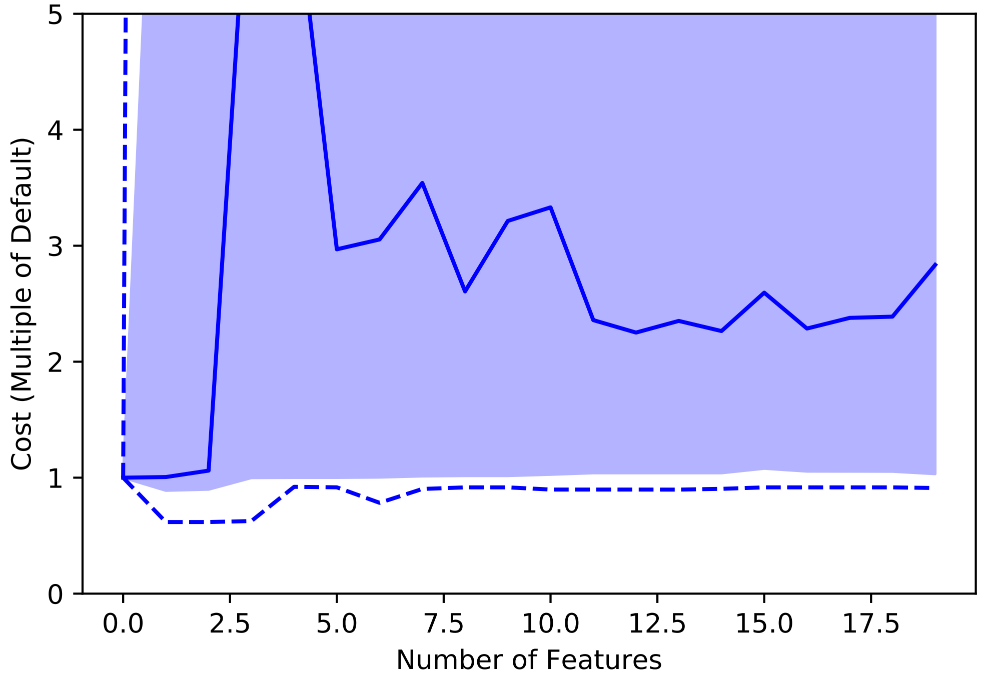
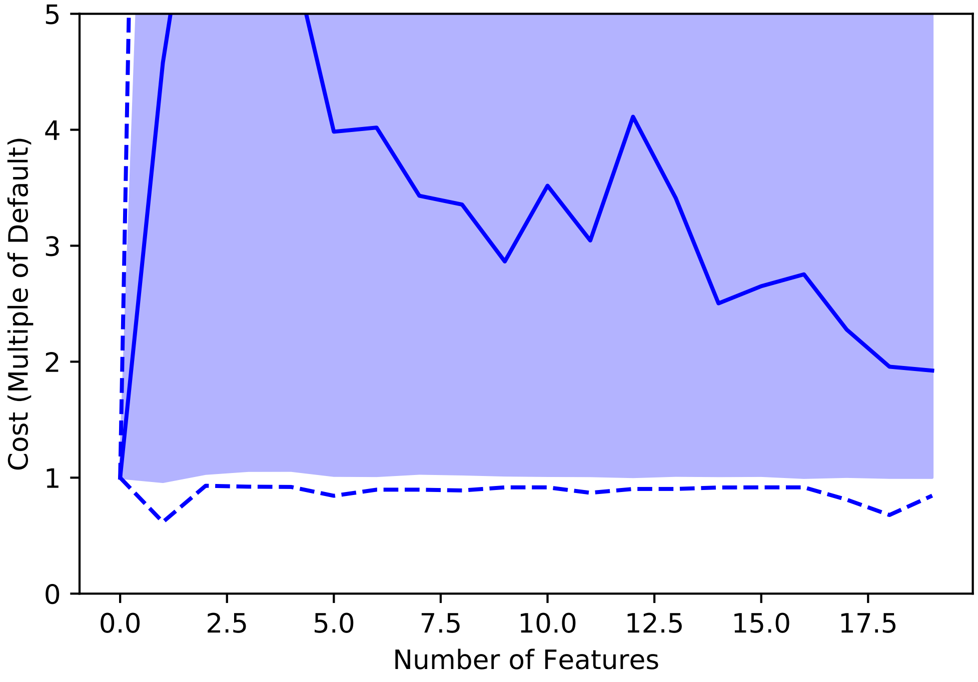


Figure 6.17: Disadvantage Linear Policy with History Summary



the policies have few features and therefore few degrees of freedom, they cannot represent policies in detail. This leads them to nearly always predict that no action should be taken, essentially falling back to the default policy and giving identical performance. Given more features, however, their performance is even worse, suggesting that they decide to collect too frequently. In very few cases do they give any improvement over the default.

6.2.3 Hellinger Tree Polices with Importance Feature Selection

We tested each of our learned policies on 100 different old sizes ranging from 1 to 5 times the maximum live size of the trace. The results from one program and input, `javac` on `asm`, are given in Figure 6.18.

Figure 6.18(b) shows the same data as Figure 6.18(a), but the cost values have been shifted and scaled to bring the cost of the optimal schedule to 0 and the cost of the default policy to 1. The graphs show tor hat the performance of the learned policy does not generalize well to small heap sizes, but for most of the range it achieves about half the possible improvement.

Unfortunately, not all traces fare as well. Figure 6.19 shows similar graphs for the large60 input to the batik program. For most of the range, the learned policy simply reproduces the default policy. For some traces, it does so for its entire range. However, the performance of the learned policies varies between inputs to a single program, not simply across different programs. Figure 6.20 illustrates a different input to the batik program on which the learned policy has much better performance than the default.

Next, Figure 6.21 gives the distribution of scaled costs for every program and input in our suite. About half the time, the learned policy is identical to the default policy, and in rare cases worse. However, in many cases the learned policy is better. It gives a mean of about 20 percent of the possible improvement so long as the heap size is not too small. In a few cases, it equals the optimal schedule.

Finally, we show a similar summary for the case where the decision trees are built using historical features as well as the features of the current time step. This is found in Figure 6.22. Note that although the polices themselves use history features, the feature selection was done only using the current time-step features. This is because the time to run the importance calculation increases linearly with the number of features that we are selecting from, and including the history features increases the number of features forty times. This makes the calculation prohibitively expensive. Instead, we select the same features as in the previous case. Unfortunately, the inclusion of history features degrades the performance of the polices, pushing the mean up to be more costly than the default policy. Historical features, at least chosen in this manner, seem to generally be less informative than the features from the current time step.

6.2.4 Hellinger Tree Polices with Mutual Information Feature Selection

Finally, we discuss the results we obtain using Hellinger tree policies, but with feature selection accomplished by mutual information algorithm. One of the primary advantages of this algorithm is that it can be executed much more quickly than the alternative that uses random-forest importance. This allows us to select from history features as well as the features of the current time step. The results are shown in Figure 6.23 for the case where only features of the current time step are selected and used. The performance is very similar to that of the default policy. In fact, the cost is not equal to that of the default for only three traces: `javac` on the `crystal` input, and `luindex` on the `default60` and `poem60` inputs.

The performance improves markedly when history features are included in the policy, as shown in Figure 6.24. These costs are quite similar to the importance case, though these policies achieve a slightly lower mean cost.

6.3 What Influences Learned Policy Performance?

It's natural to ask what aspects of a trace influence the performance of the learned policies. This may help us to predict whether a learned policy will be effective in a particular situation or further refine our policies to a particular need. In this section, we restrict our attention to results obtained using Hellinger tree policies with mutual information feature selection and history features simply because they had the best results of all of our learned policies.

First, are the performance gains limited only to certain programs? The programs differ widely in the number of inputs we have for them, but Figure 6.25 suggests that the costs vary across inputs to each program. The programs generally show a range of different values for the learned policy costs, and they tend to have many results near the default. The one possible exception is the `luindex` program, though with only a few inputs it is hard to draw firm conclusions.

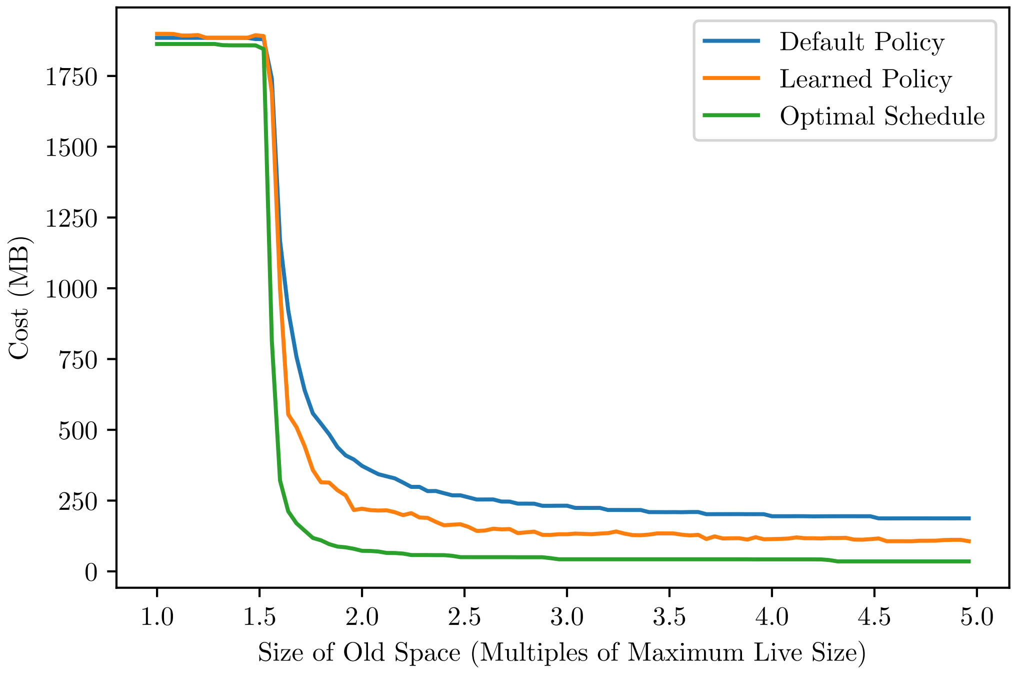
Next, we investigate whether the length of the trace influences the cost of the learned policy. The results are shown in Figure 6.26; note the logarithmic horizontal axis. Each point on the plot represents the cost of the learned policy at at a heap size of three times the maximum live size, the same size at which it is trained. The trend line has an r-value of 0.50 and a p-value of 8.3×10^{-8}, so we conclude that there is a correlation between these two values. The learned policy is able to achieve better results on shorter traces. When traces are short, each collection decision has a greater impact on the overall cost. If the policies are able to learn just a limited number of very good collection decisions, this may explain the trend we see here.

Finally, we investigate a relationship between the performance possible with the optimal schedule and the performance actually achieved by the learned policy. In other words, can the learned policy perform well only when the optimal schedule is much better than the default? Figure 6.27 answers this question in the negative. The regression line has an r-value of 0.34 and a p-value of 0.01, so we cannot conclude that any trend exists. The learned policy cost does seem to depend on whether the optimal schedule makes a large or small improvement over the default policy.

Next, in our final chapter, we summarize our conclusions and suggest ways this work may be extended in the future.

Figure 6.18: Policy costs for javac asm.

(a) Costs have not been scaled.



(b) Costs scaled to fractions of default policy cost.

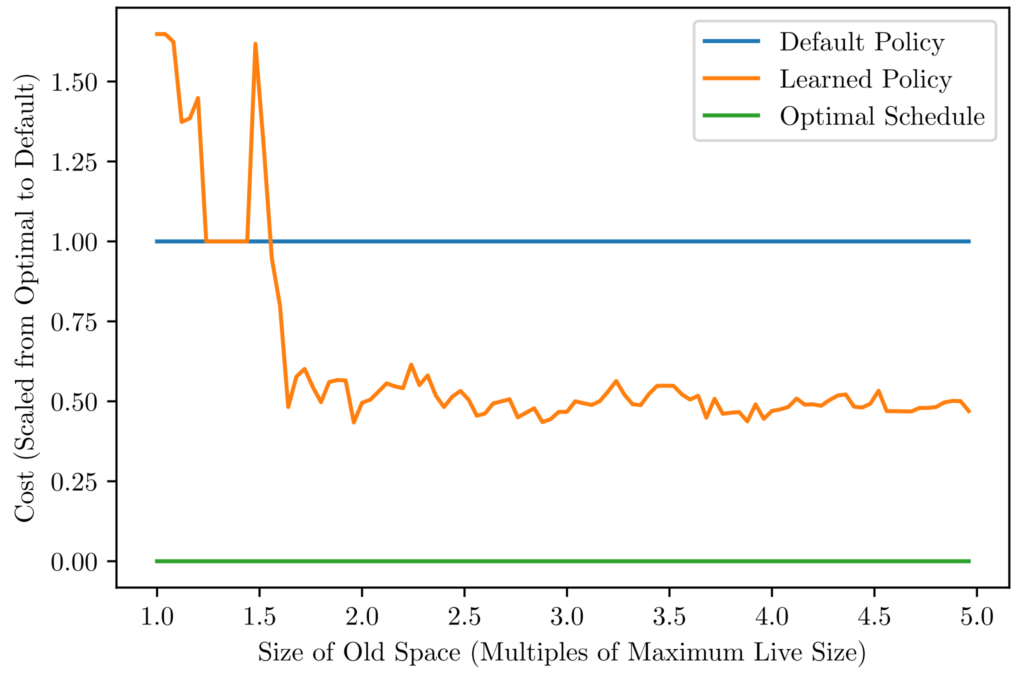
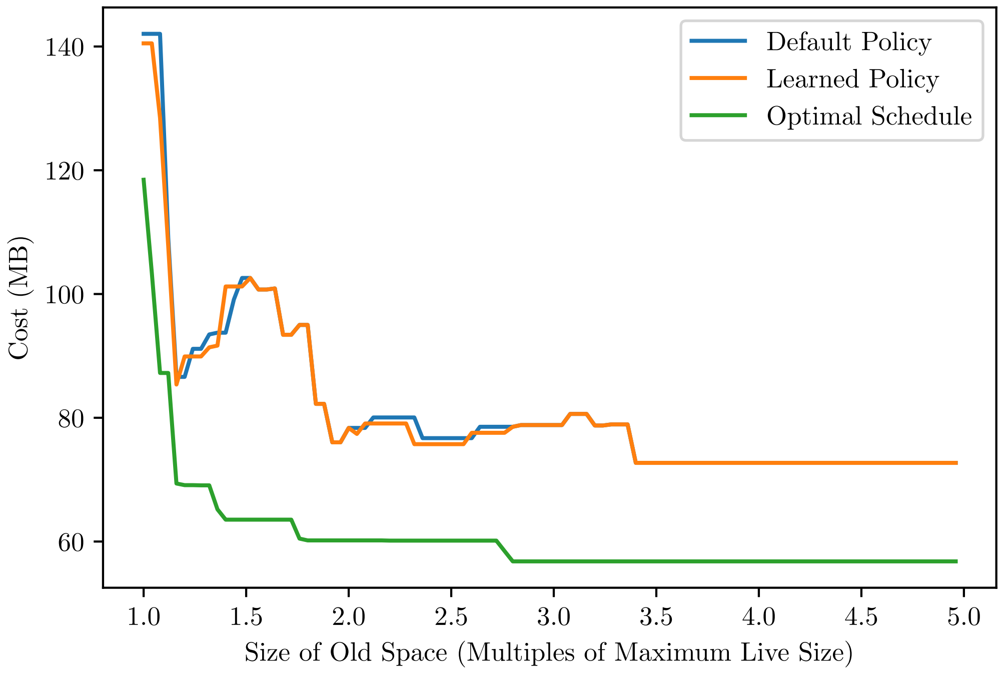


Figure 6.19: Policy costs for batik default60.

(a) Costs have not been scaled.



(b) Costs scaled to fractions of the default policy costs.

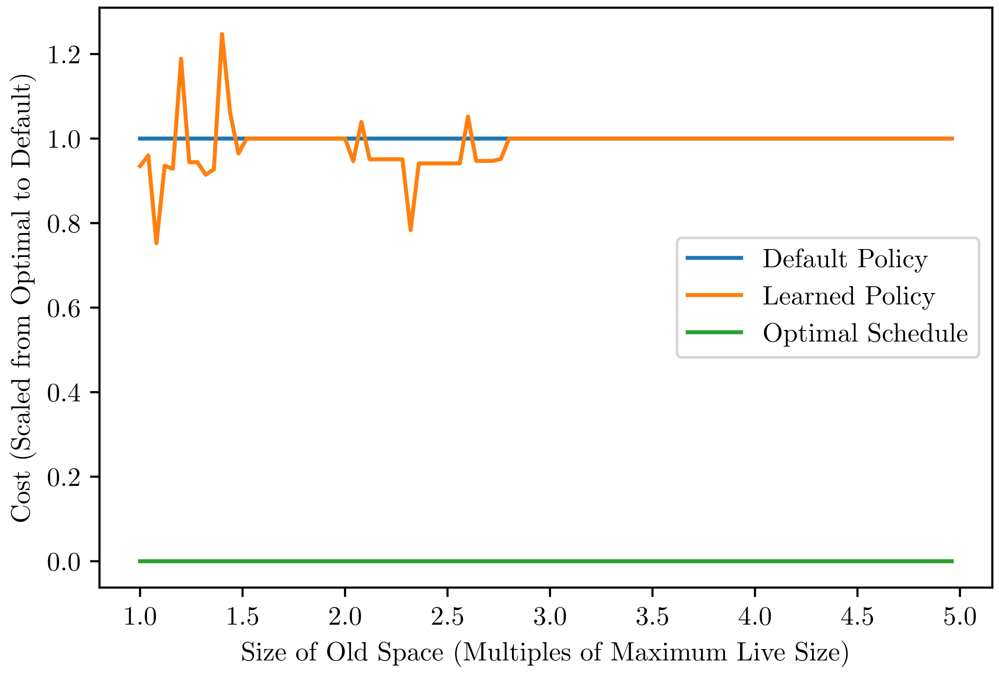
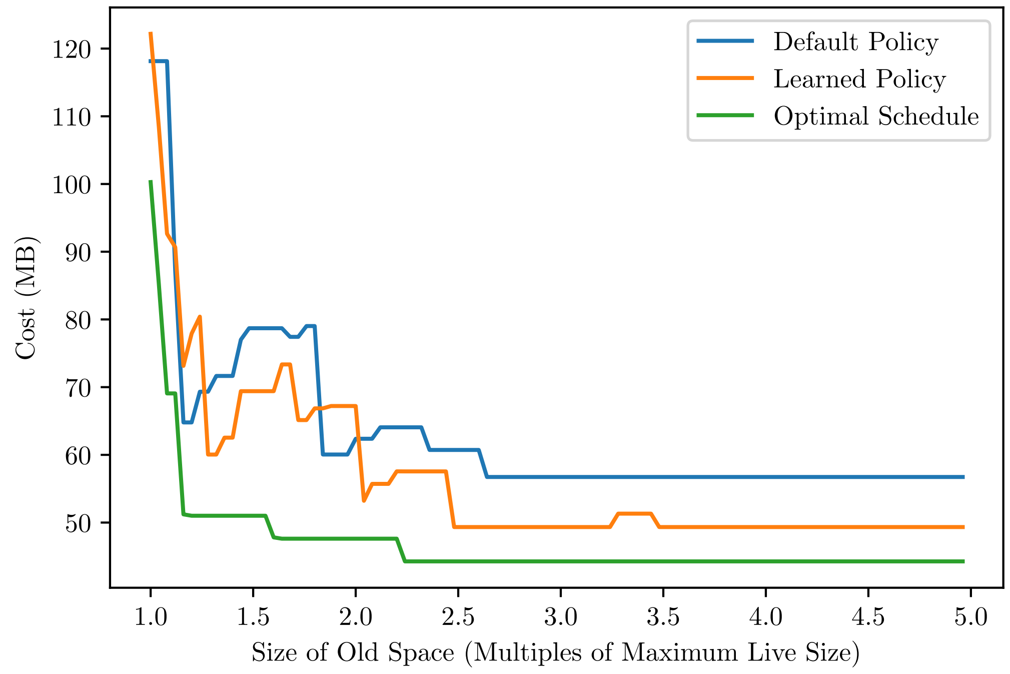
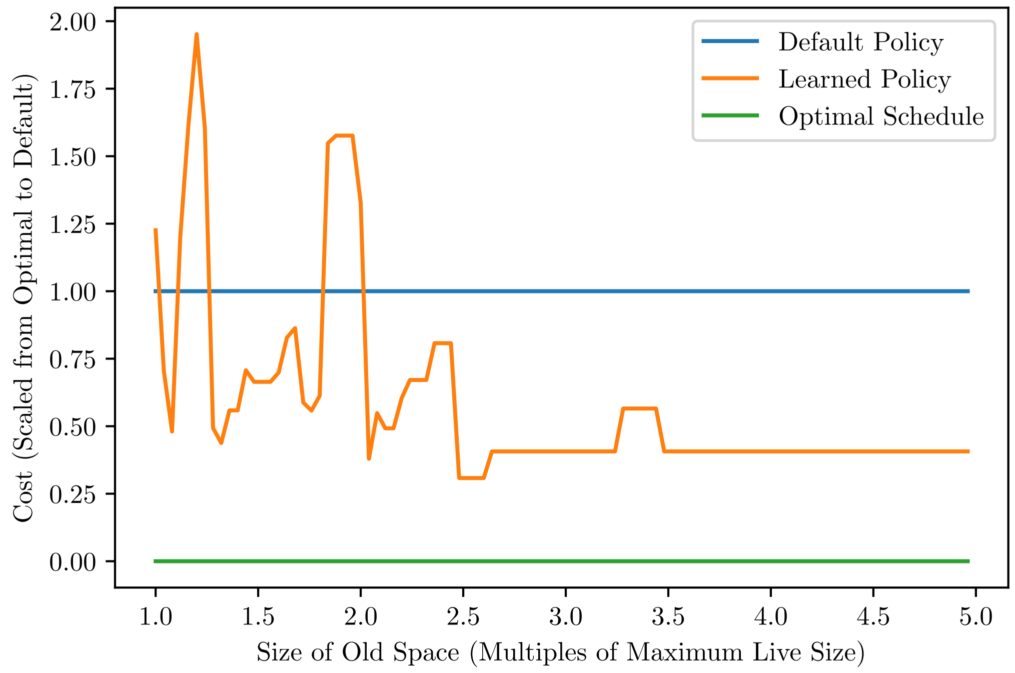


Figure 6.20: Policy costs for batik default60.

(a) Costs have not been scaled.



(b) Costs scaled to fractions of the default policy costs.



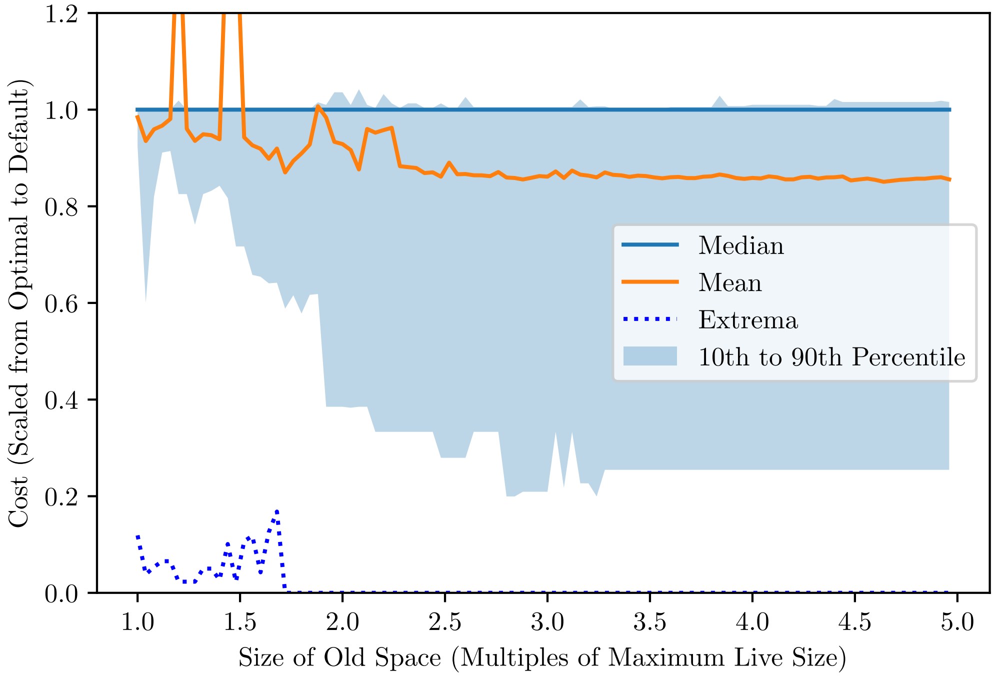


Figure 6.21: Distribution of learned policy costs.

Figure 6.22: Distribution of Hellinger tree policy costs, using importance and history features.

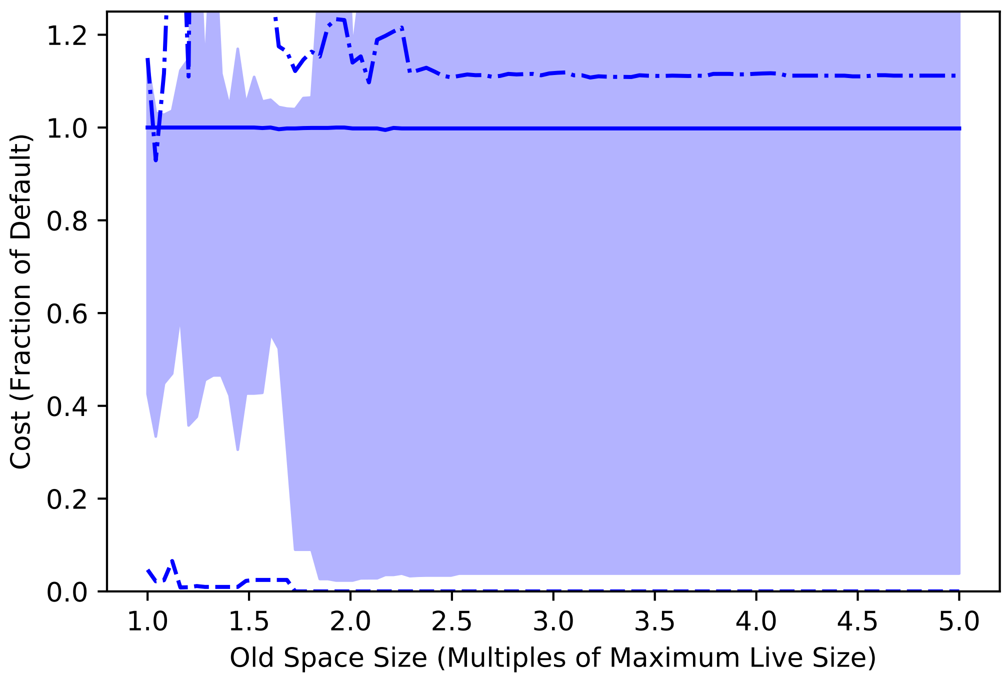


Figure 6.23: Distribution of Hellinger tree policy costs, using mutual information

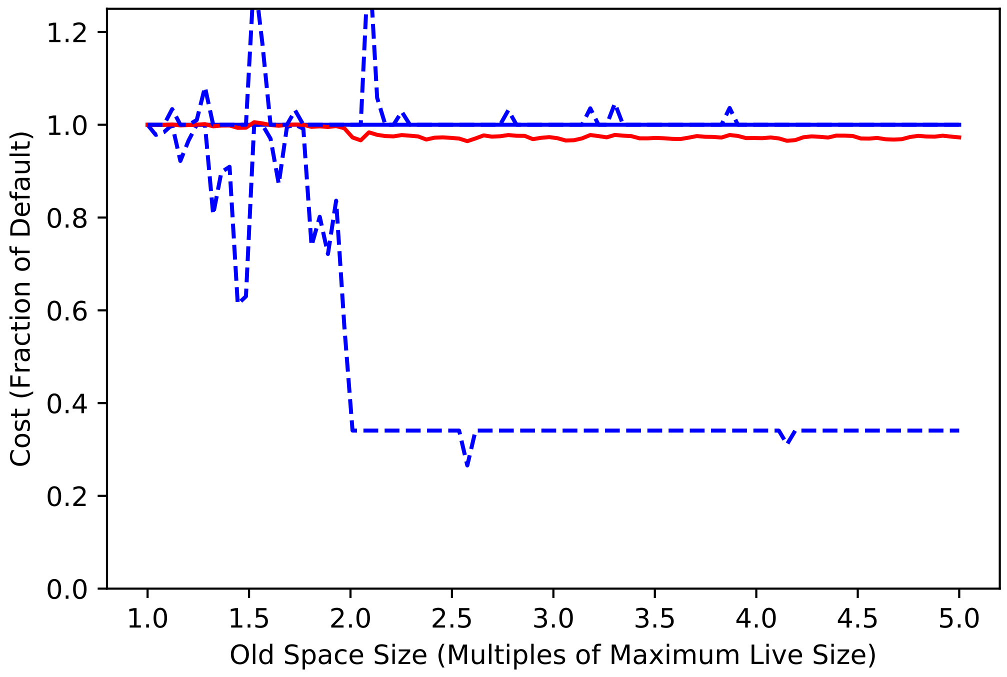


Figure 6.24: Distribution of Hellinger tree policy costs, using mutual information and history features.

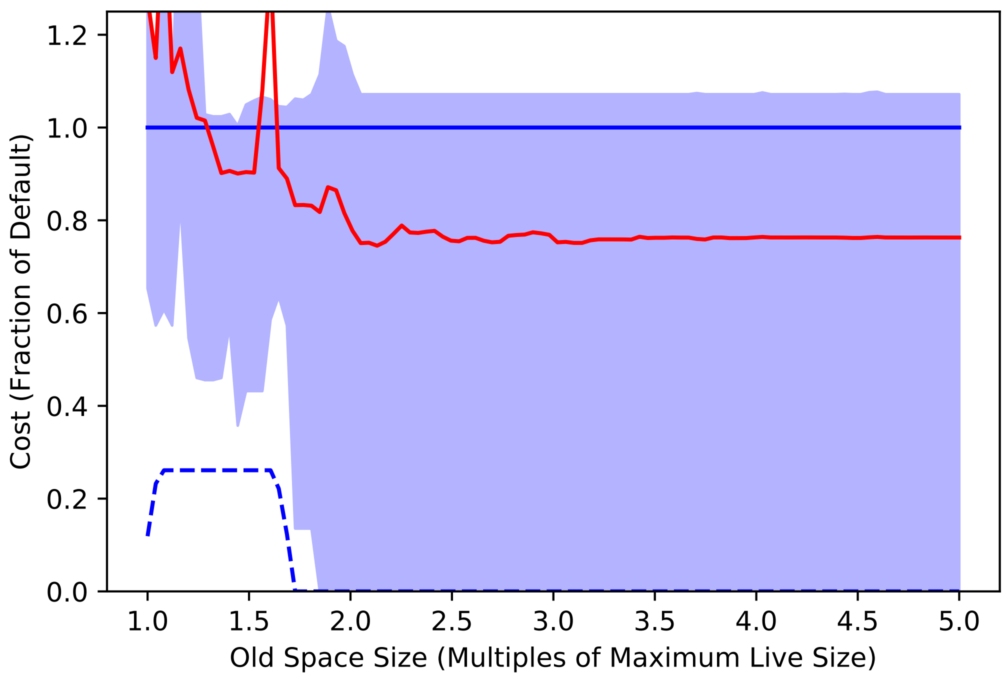


Figure 6.25: Distribution of learned policy performance across different inputs to the programs.

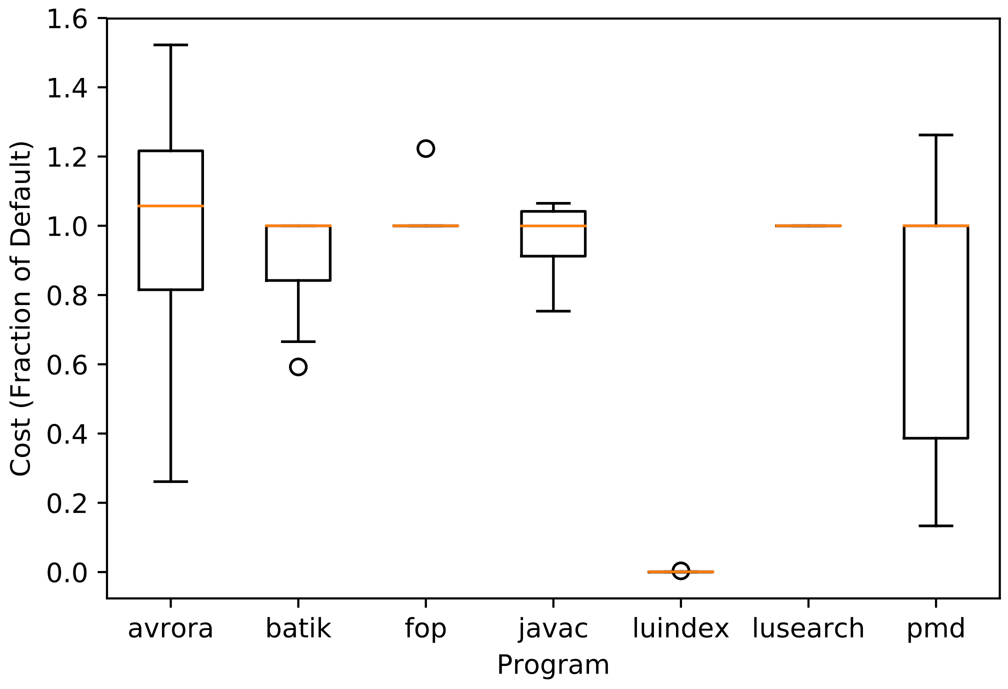


Figure 6.26: Learned policy cost vs trace length

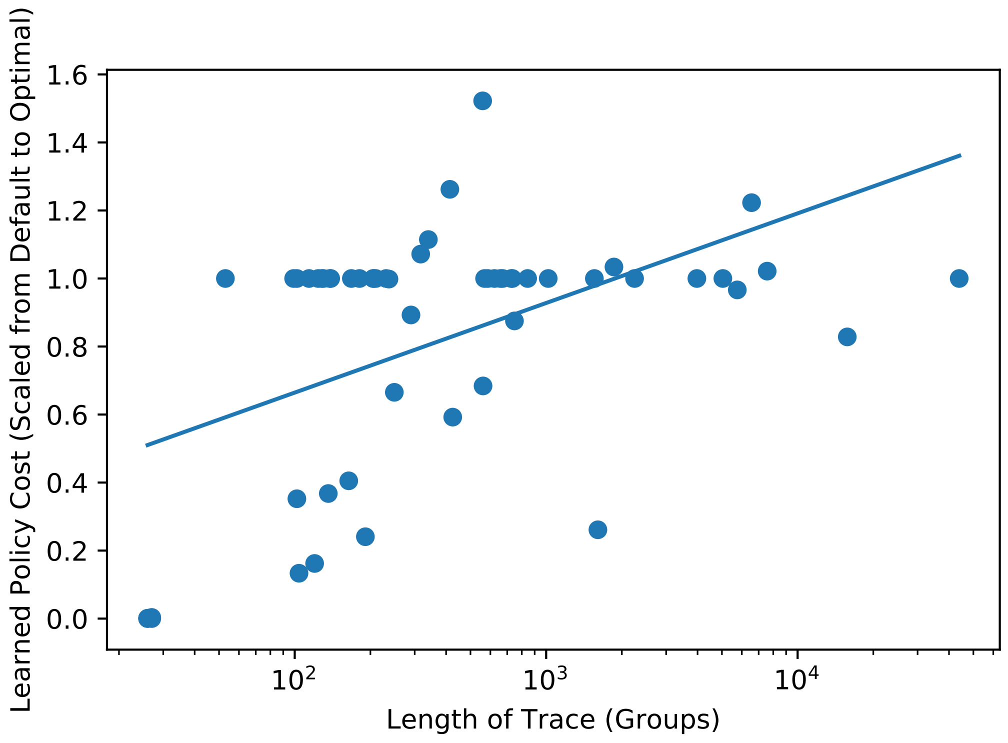
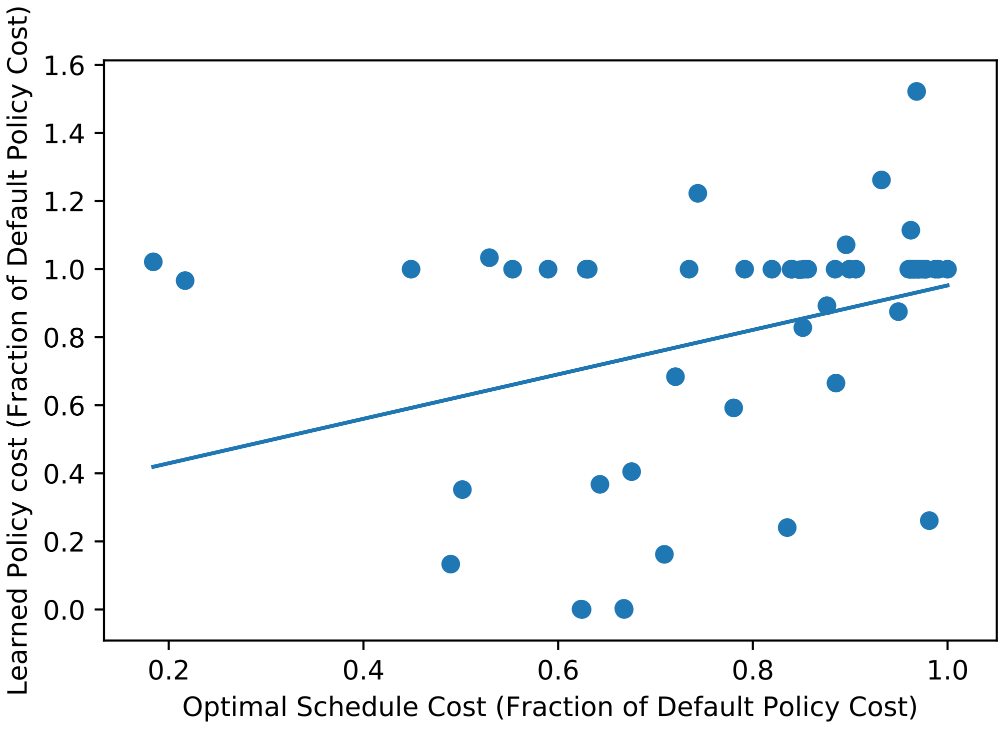


Figure 6.27: Learned policy cost vs optimal schedule cost.



CHAPTER 7

CONCLUSIONS AND FUTURE WORK

We have presented a dynamic program that can exactly calculate optimal schedules of collections, and shown that for many programs and inputs there is substantial room for improvement over the default policy. Future work might look to develop similar algorithms for different collector models. These could include different configurations of generations, or different cost calculations.

We have also shown that it is possible to learn GC policies that improve collector performance even on new inputs. To our knowledge, this is the first work in the literature to apply machine learning techniques to the problem of optimizing generational garbage collection times. Linear policies cannot adequately represent the needed complexity, and generally cannot improve over the default. Decision tree policies, on the other hand, often improve over the default and sometimes match the optimal schedule. Among the feature selection algorithms we studied, mutual information is slightly better when history features are used, and significantly worse when they are not. We can conclude that history features give some benefit, but to do so they must be considered during feature selection step as well during policy training. Simply adding the history of previously selected features can be counterproductive.

A number of avenues remain open for future research:

- There are different performance costs to instrumenting different features. A practical policy may benefit from selecting features in a cost-aware manner.
- Classifiers other than random forests may give better performance on this task. For example, our random forests classify each time step on its own and do not make

any use of the fact that our data form ordered series. Likewise, it is possible that neural net models, for example, might do better (although a quick check suggested that in this simple case they did worse).

- Our continuous features could be discretized in different manners, such as logarithmically into percentiles.
- For each program and input, our classifiers are trained on a single trace using a single heap size. Including additional heap sizes in the training data may help the learned policies to generalize to new inputs and heap sizes. This may address the issue of the sometimes bad performance for small heap sizes.

The policies are also given only the exactly optimal schedules as ground truth, but many schedules have costs only slightly higher. Training on these slightly sub-optimal schedules as well may improve the performance of our policies by offering more training data and avoiding over-sensitivity to the exact decisions needed to achieve optimal cost.

Much work remains before collection policies similar to those we investigate here could be put into production in real run-time systems. We have, however, made the important first steps toward using machine learning techniques to decide when generational garbage collectors should be run in order to optimize their performance, and we have shown that further investigation may be worthwhile.

BIBLIOGRAPHY

Amaldi, Edoardo and Viggo Kann (1998). "On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems". In: *Theoretical Computer Science* 209.1-2, pp. 237–260.

Andreasson, Eva, Frank Hoffmann, and Olof Lindholm (2002). "To collect or not to collect? Machine learning for memory management." In: *2nd Java Virtual Machine Research and Technology Symposium.* USENIX, pp. 27–39.

Archer, Kellie J and Ryan V Kimes (2008). "Empirical characterization of random forest variable importance measures". In: *Computational Statistics and Data Analysis* 52.4, pp. 2249–2260.

Bach, Francis et al. (2012). "Optimization with sparsity-inducing penalties". In: *Foundations and Trends® in Machine Learning* 4.1, pp. 1–106.

Blackburn, Stephen M. et al. (2006). "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006.* Ed. by Peri L. Tarr and William R. Cook. Portland, OR: ACM, pp. 169–190. ISBN: 1-59593-348-4. DOI: 10.1145/1167473.1167488. URL: http://doi.acm.org/10.1145/1167473.1167488.

Breiman, Leo (2001). "Random forests". In: *Machine Learning* 45.1, pp. 5–32.

Chernova, Sonia and Manuela Veloso (2007). "Confidence-based policy learning from demonstration using gaussian mixture models". In: *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems.* ACM, p. 233.

— (2008). "Teaching multi-robot coordination using demonstration of communication and state sharing". In: *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3.* International Foundation for Autonomous Agents and Multiagent Systems, pp. 1183–1186.

Cieslak, David A et al. (2012). "Hellinger distance decision trees are robust and skew-insensitive". In: *Data Mining and Knowledge Discovery* 24.1, pp. 136–158.

Cortes, Corinna and Vladimir Vapnik (1995). "Support-vector networks". In: *Machine learning* 20.3, pp. 273–297.

Fleuret, François (2004). "Fast binary feature selection with conditional mutual information". In: *Journal of Machine learning research* 5.Nov, pp. 1531–1555.

Forman, George (2003). "An extensive empirical study of feature selection metrics for text classification". In: *Journal of machine learning research* 3.Mar, pp. 1289–1305.

Guyon, Isabelle and André Elisseeff (2003). "An introduction to variable and feature selection". In: *Journal of machine learning research* 3.Mar, pp. 1157–1182.

Hertz, Matthew and Emery D Berger (2005). "Quantifying the performance of garbage collection vs. explicit memory management". In: *ACM SIGPLAN Notices*. Vol. 40. 10. ACM, pp. 313–326.

Hertz, Matthew, Stephen M. Blackburn, et al. (2006). "Generating object lifetime traces with Merlin". In: *ACM Transactions on Programming Languages and Systems* 28.3, pp. 476–516.

Inoue, Tetsunari Inamura Masayuki Inaba Hirochika, M Inamura, and H Inaba (1999). "Acquisition of probabilistic behavior decision model based on the interactive teaching method". In: *Proceedings of the Ninth International Conference on Advanced Robotics, ICAR99*.

Jacek, Nicholas, Meng-Chieh Chiu, Benjamin M. Marlin, et al. (Jan. 2019). "Optimal Choice of When to Garbage Collect". In: *ACM Trans. Program. Lang. Syst.* 41.1, 3:1–3:35. ISSN: 0164-0925. URL: http://doi.acm.org/10.1145/3282438.

Jacek, Nicholas, Meng-Chieh Chiu, Benjamin Marlin, et al. (2016). "Assessing the Limits of Program-specific Garbage Collection Performance". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: ACM, pp. 584–598. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908120.

Kohavi, Ron and George H John (1997). "Wrappers for feature subset selection". In: *Artificial intelligence* 97.1-2, pp. 273–324.

Mao, Feng, Eddy Z Zhang, and Xipeng Shen (2009). "Influence of program inputs on the selection of garbage collectors". In: *2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, pp. 91–100.

Ricci, Nathan P., Samuel Z. Guyer, and J. Eliot B. Moss (Aug. 2011). "Tool Demonstration: Elephant Tracks—Generating Program Traces with Object Death Records". In: *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*. ACM. Kongens Lyngby, Denmark: ACM, pp. 39–43.

Ricci, Nathan P., Samuel Z. Guyer, and J. Eliot B. Moss (2013). “Elephant Tracks: Portable production of complete and precise GC traces”. In: *International Symposium on Memory Management, ISMM ’13, June 20, 2013.* Ed. by Perry Cheng and Erez Petrank. Seattle, WA: ACM, pp. 109–118. ISBN: 978-1-4503-2100-6. DOI: 10.1145/2464157.2466484. URL: http://doi.acm.org/10.1145/2464157.2466484.

Ross, Stephane and J Andrew Bagnell (2014). “Reinforcement and imitation learning via interactive no-regret learning”. In: *arXiv preprint arXiv:1406.5979.*

Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell (2011). “A reduction of imitation learning and structured prediction to no-regret online learning”. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635.

Sammut, Claude et al. (1992). “Learning to fly”. In: *Machine Learning Proceedings 1992.* Elsevier, pp. 385–393.

Saunders, Joe, Chrystopher L Nehaniv, and Kerstin Dautenhahn (2006). “Teaching robots by moulding behavior and scaffolding the environment”. In: *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction.* ACM, pp. 118–125.

Schaal, Stefan (1997). “Learning from demonstration”. In: *Advances in neural information processing systems*, pp. 1040–1046.

Singer, Jeremy, Gavin Brown, Mikel Luján, et al. (2007). “Towards intelligent analysis techniques for object pretenuring”. In: *5th International Symposium on Principles and Practice of Programming in Java.* ACM, pp. 203–208.

Singer, Jeremy, Gavin Brown, Ian Watson, et al. (2007). “Intelligent selection of application-specific garbage collectors”. In: *6th ACM SIGPLAN International Symposium on Memory Management.* ACM, pp. 91–102.

Singer, Jeremy, George Kovoor, et al. (2011). “Garbage collection auto-tuning for Java MapReduce on multi-cores”. In: 46.11, pp. 109–118.

Swirszcz, Grzegorz, Naoki Abe, and Aurelie C Lozano (2009). “Grouped orthogonal matching pursuit for variable selection and prediction”. In: *Advances in Neural Information Processing Systems*, pp. 1150–1158.

Tiwari, Hemant and Vanraj Vala (2017). “Adaptive SSP forecast and memory reclamation using belief nets”. In: *2017 IEEE 4th International Conference on Soft Computing and Machine Intelligence.* IEEE, pp. 173–177.

Ungar, David (1984). “Generation scavenging: A non-disruptive high performance storage reclamation algorithm”. In: *ACM SIGSOFT Software Engineering Notes.* Vol. 9. 3. ACM, pp. 157–167.

Weston, Jason et al. (2003). “Use of the zero-norm with linear models and kernel methods”. In: *Journal of machine learning research* 3.Mar, pp. 1439–1461.

White, David R et al. (2013). “Control theory for principled heap sizing”. In: *12th ACM SIGPLAN International Symposium on Memory Management* 48.11, pp. 27–38.

Ziegler, John G and Nathaniel B Nichols (1942). “Optimum settings for automatic controllers”. In: *Transactions of the ASME* 64.11.