Learning to Optimize: from Theory to Practice

Thesis by
Jialin Song

In Partial Fulfillment of the Requirements for the
Degree of
Doctor of Philosophy

Caltech

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2021
Defended May 26, 2021

ACKNOWLEDGEMENTS

First and foremost, I want to thank my advisor, Yisong Yue. This thesis would not exist without your extraordinary guidance. Looking back at the almost six-year journey, I am amazed at how far I have come as a researcher and I want to say a sincere thank you. I have learned to love research from our countless one-on-one meetings where your enthusiasm for research shines. I have also learned many important life lessons from you. This journey did not start smoothly but your encouragement and belief in me every step of the way kept me going. I thank you for being the best advisor any student could dream of having.

I want to thank my thesis committee members, Bistra Dilkina, Richard Murray, and Adam Wierman, for taking the time to provide feedback and discussions to improve my research.

I am privileged to have worked with a long list of talented collaborators who taught me much over the years (in no particular order): Ravi Lanka, Albert Zhao, Aadylot Bhatnagar, Masahiro Ono, Bistra Dilkina, Aaron Ferber, James Bowden, Yuxin Chen, Thomas Desautels, Ayya Alieva, Aiden Aceves, Stephen Mayo, Neil Abcouwer, Shreyansh Daftry, Siddarth Venkatraman, Tyler del Sesto, Olivier Toupet, Fengze Xie, Marcus Dominguez-Kuhne, Benjamin Riviere, Wolfgang Hoenig, Soon-Jo Chung, Sihui Dai, Yury Tokpanov, Dagny Fleischman, Kate Fountaine, Harry Atwater, Curtis Hawthorne, Erich Elsen, Adam Roberts, Ian Simon, Colin Raffel, Jesse Engel, Sageev Oore, Douglas Eck, Joe Wenjie Jiang, Amir Yazdanbakhsh, Ebrahim Songhori, Anna Goldie, Navdeep Jaitly and Azalia Mirhoseini. I want to especially thank Ravi Lanka and Masahiro Ono for your hard work. I am truly lucky to have worked with you over the majority of my Ph.D. journey. It is always exciting to hear about the amazing things happening at JPL.

The CMS department provided a great curriculum. I enjoyed classes by Joel Tropp, Venkat Chandrasekaran, Yisong Yue, Adam Wierman, Thomas Vidick, Houman Owhadi, Anima Anandkumar, and Pietro Perona.

My research would not have happened without the generous support from various funding sources. The works in this thesis were funded in part by NSF awards #1637598, #1645832, #1763108, NIH #T32GM112592, and support from JPL, DARPA, DHS Center of Excellence "Critical Infrastructure Resilience Institute", Microsoft, the Rosen Bioengineering Center, Raytheon, Northrop Grumman, Beyond Limits, and UChicago CDAC via a JTFI AI + Science Grant.

During my Ph.D., I spent two summers interning at Google and I grew under the guidance of my mentors, Curtis (Fjord) Hawthorne, Doug Eck, Azalia Mirhoseini, and Navdeep Jaitly. Thanks for showing me the ropes of doing research in the industry and patiently answering my endless questions.

I also need to thank people from the Caltech community who helped me along the way. I learned a lot from Stephan Zheng, Hoang Le, Eric Zhan, and Yuxin Chen when I started out doing research. The Yue group have always been a source of inspiration and I am grateful to explore the fascinating world of machine learning with you.

I am lucky to share the office with awesome people such as Sumanth, Juba, Hao, and Rachel. I will cherish the memories of our laughter as well as our despair. Thank you to my friends De, Gautam, Navid, Florian, Mandy, Jennifer, Richard, Xinying, Karena, Yu, Jenish, Andrea, Kamyar, John, and many more for all the good time. You bring joy to my life. I especially want to thank De for being an awesome roommate during the craziness of COVID-19.

Thank you to the always helpful CMS admins, Maria, Sidney, Diane, Carmen, and Sheila, and CMS system admins David and Patrick, for tolerating my forgetting passwords way too often. Thanks to Laura and Daniel from ISP for your help navigating the international student life.

I want to thank my chamber music coaches Maia, Delores, and Robert for enriching my life with music. It has been a great pleasure to escape to music when I get stuck on research.

Finally, I want to thank my parents whose unconditional love and unwavering support have always accompanied me. Our weekly video chat is always wonderful and I am extremely grateful to have you in my life. You are the source of energy for me to keep pushing forward and I hope I make you proud.

ABSTRACT

Optimization is at the heart of everyday applications, from finding the fastest route for navigation to designing efficient drugs for diseases. The study of optimization algorithms has focused on developing general approaches that do not adapt to specific problem instances. While they enjoy wide applicability, they forgo the potentially useful information embedded in the structure of an instance. Furthermore, as new optimization problems appear, the algorithm development process relies heavily on domain expertise to identify special properties and design methods to exploit them. Such design philosophy is labor-intensive and difficult to deploy efficiently to a broad range of domain-specific optimization problems, which are becoming ubiquitous in the pursuit of ever more personalized applications.

In this dissertation, we consider different hybrid versions of classical optimization algorithms with data-driven techniques. We aim to equip classical algorithms with the ability to adapt their behaviors on the fly based on specific problem instances. A common theme in our approaches is to train the data-driven components on a pre-collected batch of representative problem instances to optimize some performance metrics, e.g., wall-clock time. Varying the integration details, we present several approaches to learning data-driven optimization modules for combinatorial optimization problems and study the corresponding fundamental research questions on policy learning. We provide multiple practical experimental results to showcase the practicality of our methods which lead to state-of-the-art performance on some classes of problems.

PUBLISHED CONTENT AND CONTRIBUTIONS

Abcouwer, Neil et al. (2021). "Machine Learning Based Path Planning for Improved Rover Navigation". In: *IEEE Aerospece Conference*. URL: `https://arxiv.org/abs/2011.06022`.
J.S. participated in the discussion of the project.

Alieva, Ayya, Aiden Aceves, Jialin Song, Stephen Mayo, Yisong Yue, and Yuxin Chen (2021). "Learning to Make Decisions via Submodular Regularization". In: *International Conference on Learning Representations*. URL: `https://openreview.net/forum?id=ac288vnG_7U`.
J.S. participated in the analysis of the algorithm and participated in the writing of the manuscript.

Ferber, Aaron, Jialin Song, Bistra Dilkina, and Yisong Yue (2021). "Learning Pseudo-Backdoors for Mixed Integer Programs". In: *Symposium on Combinatorial Search*.
J.S. participated in the conception of the project, conducted experiments jointly with Aaron Ferber and participated in the writing of the manuscript.

Song, Jialin, Ravi Lanka, Yisong Yue, and Bistra Dilkina (2020). "A General Large Neighborhood Search Framework for Solving Integer Linear Programs". In: *Advances in Neural Information Processing Systems*. Vol. 33. URL: `https://proceedings.neurips.cc/paper/2020/hash/e769e03a9d329b2e864b4bf4ff54ff39-Abstract.html`.
J.S. participated in the conception of the project, designed the algorithm, conducted experiments jointly with Ravi Lanka, and participated in the writing of the manuscript.

Song, Jialin, Ravi Lanka, Yisong Yue, and Masahiro Ono (2020). "Co-training for policy learning". In: *Uncertainty in Artificial Intelligence*. PMLR, pp. 1191–1201. URL: `http://proceedings.mlr.press/v115/song20b.html`.
J.S. participated in the conception of the project, designed and analyzed the algorithm, conducted experiments jointly with Ravi Lanka, and participated in the writing of the manuscript.

Song, Jialin, Ravi Lanka, Albert Zhao, Aadyot Bhatnagar, Yisong Yue, and Masahiro Ono (2018). "Learning to search via retrospective imitation". In: *arXiv preprint*. URL: `https://arxiv.org/abs/1804.00846`.
J.S. participated in the conception of the project, designed and analyzed the algorithm, conducted experiments jointly with Ravi Lanka, and participated in the writing of the manuscript.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Number *Page*

Chapter 1

INTRODUCTION

1.1 Motivation

Optimization problems are at the heart of everyday applications, from finding the fastest route to a destination to designing more effective drugs. The desire to searching for better solutions is what drives human civilization forward. The study of algorithm design for optimization problems spans areas in computer science and mathematics. Traditionally, the design of a novel algorithm calls for a similar level of creativity as in the production of a mathematical proof. An algorithm designer must study special properties of a type of optimization problem, e.g., convexity or submodularity, then create an algorithm that exploits such properties. Both identifying properties and making use of them take considerable expertise and can potentially take a long time. The field of machine learning studies a relevant problem: can an algorithm / a program be learned from data? Indeed, deep learning has revolutionized areas such as computer vision. Instead of asking a human to write an algorithm to label whether an image contains a cat, deep learning models learn weights of deep neural networks from a large amount of labeled cat and no-cat images. The result is a model, acting as a program, able to generalize to future images. This is a fundamental perspective shift about how to solve problems in the presence of large amounts of data: from manual design algorithms according to problem descriptions, to automatic discovery of generalizable patterns from data.

Can we incorporate machine learning into the algorithm design process? This is a broad question and this thesis provides several concrete approaches to combining learned models with standard algorithms. However, as a first step, we need to consider the question of whether data-driven approaches can help. On closer inspection of general optimization algorithms, we can break an algorithm into two parts. Firstly, there is the outline of the algorithm which describes the steps to follow to arrive at a final solution. This part is typically fixed independent of particular problem instances. Secondly, some parameters in the outline control the behavior of an algorithm in the optimization process for individual instances. For example, consider the simple gradient descent algorithm for minimizing a differentiable function f, where the algorithmic outline is the gradient update step $x_{n+1} = x_n - \alpha_n \nabla f(x_n)$

which is applied repeatedly. The parameter is the step size $\alpha_n \in \mathbb{R}^+$, which is crucial for convergence of the algorithm (Boyd and Vandenberghe, 2004). While there are theoretical results on what the step size should be (or be proportional to), setting the actual value for each problem is still largely an exercise in intuition. Andrychowicz et al. (2016) formulate setting learning rate as a learning problem and apply recurrent neural networks to adaptively decide a learning rate for each step.

Another example where the parameters are crucial is the heuristic function to estimate cost-to-go in the A* algorithm (Hart, Nilsson, and Raphael, 1968). Dechter and Pearl (1985) provide analysis on the consistent and admissible properties of a heuristic function. However, consistency and admissibility only provide constraints that a heuristic function needs to satisfy and there are usually many feasible choices. In practice, simple heuristics such as the Euclidean distance or the Manhattan distance are commonly used, both of which are based on hand-crafted formulas. A data-driven way differs in replacing heuristics with machine learning models to fully exploit the structure of individual problem instances, as we have demonstrated in (J. Song, Lanka, Zhao, et al., 2018).

The learning perspective favors adaptive models over hand-crafted rules for setting those parameters. With an expressive enough model class, such as deep neural networks, we are likely to be able to find models that outperform existing approaches. Is there any downside to using a learned model? Algorithm design has traditionally placed great emphasis on the correctness of an algorithm. For example, a large portion of the popular introductory algorithm book (Cormen et al., 2009) is dedicated to proofs. Hand-crafted heuristics are nice for proofs: they usually have simple forms and their properties are easily understandable. In contrast, a deep neural network, which is notorious for its opaqueness, does not play well with theoretical analyses. As a result, we need to trade-off guarantees with performances. That is not to say that we have to sacrifice all the nice theoretical properties. Careful designs can result in data-driven algorithm with guarantees (Balcan et al., 2018; B. Sun et al., 2020). We will show in later chapters how to design learning algorithms that maintain performance guarantees as well.

How to design a data-driven algorithm is the focus of this thesis. Generally speaking, a data-driven approach can be characterized by the ratio of the data-driven component to the traditional algorithm outline. One could discard all existing algorithm structures and replace them with a machine learning model, as shown in the recent works on using graph neural networks to solve the traveling salesman problem (Kool, Hoof, and Welling, 2018; Joshi, Laurent, and Bresson, 2019). Of course, one could adopt a more gentle approach and try to fuse machine learning models into an existing algorithmic framework. In this thesis, we explore various ways to realize the latter perspective.

Fundamentally, an optimization algorithm solves a search problem: from a feasible set of solutions, we want to find one that optimizes some objective function. Depending on problem types, algorithms are designed to either evolve solutions iteratively or build up a complete solution through a sequential process. For example, if the objective function is convex and differentiable, we can sequentially update the solution by performing gradient descent until convergence to an optimum. Another example is solving integer linear programs (ILPs) with the branch-and-bound (BnB) algorithm (Lawler and Wood, 1966). The search domain for an ILP is combinatorial due to the presence of integer variables. The BnB algorithm builds a search tree to navigate the search space sequentially.

Two important metrics to evaluate the quality of an optimization algorithm are speed and solution quality. The speed reflects how fast an algorithm terminates and the quality measures how good the solution at termination is. To achieve better performance, we need to make good decisions on navigating the search space, which often means that the search algorithm needs to dynamically adapt its behavior based on history so that future efforts are spent on promising areas. Because of the sequential nature of the search, we model general optimization problems as sequential decision-making problems. The goal is to learn policies, which map some featurization of states to actions, that result in good performance metrics. The benefit of this approach is to reduce the dependency on domain experts and arrive at more adaptive heuristics, as such a modeling approach allows more fine-grained control on their behaviors.

1.2 Challenges

With the motivation clear, we now describe a common pipeline for learning to optimize. In practical applications, we are often concerned with solving a narrow collection of optimization problems. For example, airlines are interested in scheduling problems while delivery services are interested in vehicle routing problems. Furthermore, there are large overlaps between new problem instances and the old ones, which is an ideal setting for applying machine learning algorithms as we know the training and testing sets are similar. Thus, the learned model can generalize. From these observations, we can set up a learning pipeline. For training, we have a collection of readily available problem instances, e.g., already solved instances. Then we will apply machine learning techniques tailored for those problems. At test time, e.g., when new instances are presented, we apply the learned models to solve them. The details in this pipeline depend on some unique properties that optimization problems have, which we must consider when designing appropriate machine learning algorithms. We briefly discuss three aspects that are the focuses of this thesis.

Interface with Existing Solvers Thanks to previous research, there are existing algorithms and solvers for widely studied optimization problems. They are often capable to solve moderately sized problems but have scalability issues. This is a major difference from other settings for sequential decision-making, such as those in robotics where existing control policies can be quite bad. We view the existing solvers as an asset for a learning method and explore ways to fuse them with learning. Based on how much modification we make to a solver, we can roughly divide learning to optimize approaches into two categories.

The first category targets a sub-routine in an optimization algorithm. For example, in the BnB tree search algorithm, there are three decision modules in the evolution of the search tree. The branching module decides which integer variable to branch on. The node selection module decides which node to explore among the frontier nodes in the current search tree. Finally, the pruning module decides when to discard an entire subtree because it will not lead to better solutions. Research has shown that these decisions significantly impact the solving time of integer programs (J. T. Linderoth and M. W. Savelsbergh, 1999; Achterberg, 2009; Le Bodic and Nemhauser, 2015). Innovating on their designs is an active research area involving a large amount of heuristics. For example, the leading commercial solver Gurobi has four branching strategies (Gurobi Optimization, 2021) based on different hand-crafted scoring functions on variables. It is natural to wonder if better strategies can adapt to both the problem structure and the search process. Machine learning models provide an attractive alternative. Existing works on learning branching rules have shown that it is possible to outperform the default ones (E. Khalil, Le Bodic, et al., 2016; Gasse et al., 2019).

Another example is the genetic algorithm, which is a popular evolutionary optimization approach that finds successful use cases from deep space scheduling for satellites (Guillaume et al., 2007) to drug design (Douguet, Thoreau, and Grassy, 2000). It consists of three components: initialization, selection, and crossover (De Jong, 2006). They are used iteratively to evolve existing solutions towards better ones. Each component can be realized with a set of heuristic rules. For example, the selection step uses a fitness function to imitate natural selection and filters out low-quality solutions. Then the crossover phase happens by combining the remaining members with heuristic rules to produce the next generation of solutions. As a result, a concrete realization has several hand-crafted heuristics. Paliwal et al. (2020) uses reinforcement learning to learn a machine learning model to perform the crossover step and shows that the resulting model outperforms default crossover rules by a large margin.

A limiting factor of this kind of approach is the requirement for access to internal components of a solver or a highly flexible API, which is typically unavailable for commercial solvers. This can be a significant disadvantage if we want to compete in terms of wall-clock runtime with state-of-the-art solvers which are often commercial, e.g., Gurobi and CPLEX for solving integer programs.

The second category takes a more holistic view. Instead of updating a sub-routine with a machine learning model, a model is tasked with modifying some global behavior of the algorithm by setting high-level control parameters. This category enjoys wider applicability as it can adapt to different solver interfaces and usually requires no access to the internals of a solver. We can divide this category further into two sub-categories. The first one is algorithm configuration (Hoos, 2011). The motivation is that a typical solver has lots of parameters we can set to alter its behavior for different problem instances. The performance difference between a good setting and a bad one is often quite dramatic. As a result, algorithm configuration aims to learn a model that maps features of a problem instance to a parameter setting which leads to good performance. A popular approach formulates the algorithm configuration task as a model-based optimization problem. A machine learning model takes a problem instance and a parameter setting as input, then outputs a prediction on a performance metric, typically the runtime. At test time, the model proposes multiple parameter settings to find the best one (Hutter, Hoos, and Leyton-Brown, 2011; Kleinberg et al., 2019).

The second sub-category uses a solver as a sub-routine in a larger solving pipeline. For example, Gomes and Selman (2001) takes advantage of the variability among different stochastic search algorithms to construct a portfolio of algorithms and explores various ways to combine running different algorithms. Machine learning techniques have also been studied in the context of SAT solving (Xu et al., 2008) which won several SAT competitions. The core idea is to learn a regression model that predicts the runtime of each solver for a SAT instance. At test time, the model chooses to run the solver with the lowest predicted runtime.

We will present learning algorithms covering all the categories and point out the advantages and disadvantages in each scenario.

Choice of Learning Algorithms Because of the diverse interfacing methods with existing solvers, there is a multitude of learning algorithm choices. As an optimization algorithm typically operates sequentially, formulating the search as a sequential decision-making problem allows us to connect with the existing policy learning literature. Our ultimate goal is to learn a policy that maps a state feature that contains information about the search history to an action that continues the search. Popular policy learning algorithms can be categorized into reinforcement learning and imitation learning.

In a reinforcement learning setting (Sutton and Barto, 2018), we have access to an environment from which we can collect reward information. An agent interacts with the environment by following a policy. Feedback from environmental rewards is used to update the policy. By parametrizing policies with deep neural networks, deep reinforcement learning has achieved impressive results from game playing (Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015; Silver et al., 2016) to robotics (Gu et al., 2017; Haarnoja et al., 2019). However, reinforcement learning algorithms are usually brittle (Henderson et al., 2018) and require millions of interactions with the environment (Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015; Silver et al., 2016). Such challenges become more prominent when positive reward feedback from the environment is sparse. In particular, for discrete optimization problems, we often do not obtain any feedback, as the objective function value, until a complete solution has been found. We often have control over the definition of the reward function. To make the learning task more tractable, we can perform reward shaping (Ng, Harada, and Russell, 1999) to mitigate the sparsity issue. For example, in continuous optimization, we could shape the reward as the difference of objectives between a new solution and an old one. This method can provide dense feedback, but in cases where good solutions are rare the rewards will be negative in most steps and they provide limited information to guide the policy learning. As a result, applying reinforcement learning successfully requires careful selection of problem domains and proper formulations. Recent successes include learning to select cutting planes for integer programming (Tang, Agrawal, and Faenza, 2020) and end-to-end models for solving traveling salesman problems (Kool, Hoof, and Welling, 2018).

Imitation learning algorithms aim to address common challenges of reinforcement learning by utilizing expert demonstrations. Similar to reinforcement learning, imitation learning has seen applications in game playing (Aytar et al., 2018) and self-driving cars (Bojarski et al., 2016). Imitation is an integral part of animal learning (Galef Jr, 1988) and its impact has been studied extensively in social and cultural settings (Heyes and Galef Jr, 1996; Nehaniv and Dautenhahn, 2007). Inspired by such observations, we can incorporate demonstrations into policy learning. The access to demonstrations reduces the need for sufficient exploration, which is the root cause of the challenges that reinforcement learning faces. However, using imitation learning for learning to optimize proves to be a challenging task itself due to the following reasons.

- The type of demonstrations available is usually generated from existing solvers, either in the form of final solutions or records of the search process. Since the goal of learning is to outperform existing solvers (the experts), simply applying existing imitation learning algorithms is not enough. Methods explicitly focusing on producing policies better than experts such as the work by Chang et al. (2015) still require large amounts of exploration. Another complicating factor is the constraint resulting from interfacing with solvers. A popular category of imitation learning algorithms gathers expert demonstrations through interactive queries, e.g., (Stéphane Ross, Gordon, and D. Bagnell, 2011; Jauhri, Celemin, and Kober, 2020). They require the ability to obtain feedback at an arbitrary state. However, limited access to the internals of optimization solvers greatly limits what states we can initialize a solver with. This means that obtaining interactive feedback is nearly impossible and poses challenges in algorithmic design. We will show our attempts to address such challenges and their empirical effectiveness.
- In some cases, it is unclear what an expert is or it is infeasible to collect demonstrations, e.g., it takes a solver too long to produce solutions. This often happens when we consider novel formulations incorporating learning

components. It might seem paradoxical to do imitation learning **without** an expert but we will show that creatively using heuristics can still produce meaningful "demonstrations". A core technique is to use weaker policies, such as random exploration or policies trained on easier problems, to jump-start the learning process and iteratively refine the learned policies.

Data Collection Considerations Data are at the heart of every machine learning application. From our discussion on the choices of learning algorithms, it is clear that the required data collection is closely tied to the chosen learning algorithm. We are faced with additional constraints for data collection in learning to optimize because running solvers on problem instances to generate data is often expensive as lots of problem classes are NP-hard in combinatorial optimization. As a result, the total data volume we can expect is typically smaller than traditional settings of policy learning where cheap simulators are available, e.g., OpenAI Gym (Brockman et al., 2016). To make matters worse, the rigidity of interfacing with solvers means that we often are unable to collect some helpful information, e.g., the interactive feedback for imitation learning. There has been a recent effort to make a more accessible interface for learning to branch in integer programs (Prouvost et al., 2020), but this challenge remains in other areas. As a result, our learning methods are usually required to work with easily collectible statistics from solvers. Some examples are final solutions, objective values, and solve time. We show that, by taking advantage of properties of optimization problems and making proper algorithm choices, we can still successfully integrate learning despite those challenges.

1.3 Thesis Organization

In this thesis, we use the three challenges outlined in the previous section as a guide for learning algorithm designs. We focus on combinatorial optimization problems and integer programs. Our study will be multi-faceted, covering the spectrum from the theoretical analyses of algorithms to practical efficacy in real-world applications. Through the chapters, we will give an account of our evolving understanding of the research area.

In chapter 2, we review related works on learning to optimize and policy learning for sequential decision-making problems.

In chapters 3 and 4, we present two novel policy learning algorithms that blend imitation learning with reinforcement learning. Their creations are tightly related to properties of combinatorial optimization and integer programs. The focus of both chapters is on updating a component of a solver with a data-driven model. In chapter 3, we propose the retrospective imitation learning algorithm (J. Song, Lanka, Zhao, et al., 2018) that can turn sparse environmental rewards into dense per-step feedback, which is ideal for reduction-based imitation learning algorithms. We show that retrospective imitation provides one answer when we are unable to collect expert demonstrations. In chapter 4, we are motivated by the observation that various representations of optimization problems provide additional possibilities on interactions between learning policies. The algorithm CoPiEr (J. Song, Lanka, Yue, and Ono, 2020), inspired by the co-training algorithm (Blum and Mitchell, 1998), applies to both reinforcement and imitation learning settings.

Then we proceed with chapter 5 to study more high-level integration of learning algorithms with existing solvers. Both ideas originated from the issue of the dependency on open-sourced solvers for internal access in the previous two chapters. While our algorithmic contributions lead to improvement over the baseline solvers, they are not competitive with state-of-the-art solvers in wall-clock time. We first study a divide-and-conquer approach for integer programs that utilize a solver as a sub-routine in the conquer step. The learned policy makes high-level decisions on how to divide a problem into several sub-problems. Next, we present a project motivated by the concept of backdoor variables (R. Williams, Gomes, and Selman, 2003; Dilkina, Gomes, and Sabharwal, 2009), which are a subset of variables that capture the core hardness of optimization problems. We learn models to identify such variables and the predictions can guide a solver by raising branching priorities in solving mixed integer programs (MIPs). This chapter is based on two papers (J. Song, Lanka, Yue, and Dilkina, 2020; Ferber, J. Song, et al., 2021).

In chapter 6, we explore two ideas that are influenced by the general philosophy of learning to optimize. We design machine learning surrogate models to guide the optimization process. In the first part, a safety cost module is learned to replace an expensive computation process in Mars rover path planning. In the second part, a surrogate regularized with a submodular-norm is learned to guide a search over a combinatorial space. Both ideas embody the flexibility of the learning to optimize framework and its potential for large practical impact. This chapter is based on two papers (Abcouwer et al., 2021; Alieva et al., 2021).

Finally, we conclude with chapter 7 where we explore promising future directions in the learning to optimize research area.

Chapter 2

RELATED WORKS

2.1 Related Works on Learning to Optimize

An increasingly popular paradigm for the automated design and tuning of solvers is to use learning-based approaches. Broadly speaking, one can categorize most existing "learning to optimize" approaches into three categories: (1) learning search heuristics such as for branch-and-bound; (2) tuning the hyperparameters of existing algorithms; (3) learning to identify key substructures that an existing solver can exploit; and (4) differentiating through an optimization algorithm. In this section, we survey these paradigms.

Learning to Search In learning to search, one typically operates within the framework of a search heuristic, and trains a local decision policy from training data. Perhaps the most popular search framework for integer programs is branch-and-bound (Land and Doig, 2010), which is a complete algorithm for solving integer programs (ILPs) to optimality. Branch-and-bound is a general framework that includes many decision points that guide the search process, which historically have been designed using carefully attained domain knowledge. To arrive at more automated approaches, a collection of recent works explore learning data-driven models to outperform manually designed heuristics, including learning for branching variable selection (E. Khalil, Le Bodic, et al., 2016; Gasse et al., 2019), or node selection (He, Daume III, and Eisner, 2014; J. Song, Lanka, Zhao, et al., 2018; J. Song, Lanka, Yue, and Ono, 2020). Moreover, one can also train a model to decide when to run primal heuristics endowed in many ILP solvers (E. B. Khalil et al., 2017). Many of these approaches are trained as policies using reinforcement or imitation learning.

Writing highly optimized software implementations is challenging, and so all previous work on learning to branch-and-bound were implemented within existing software frameworks that admit interfaces for custom functions. The most common choice is the open-source solver SCIP (Achterberg, 2009), while some previous work relied on callback methods with CPlex (Bliek1ú, Bonami, and Lodi, 2014; E. Khalil, Le Bodic, et al., 2016). However, in general, one cannot depend on highly optimized solvers being amenable to incorporating learned decision procedures as subroutines. For instance, Gurobi, the leading commercial ILP solver according to (Mittelmann, 2017; Optimization, 2019), has very limited interface capabilities, and to date, none of the learned branch-and-bound implementations can reliably outperform Gurobi in wall-clock time.

Beyond branch-and-bound, other search frameworks that are amenable to data-driven design include A* search (J. Song, Lanka, Zhao, et al., 2018), direct forward search (E. Khalil, H. Dai, et al., 2017), path planning (Ichter, Harrison, and Pavone, 2018; B. Chen, B. Dai, and L. Song, 2020; T. Huang, Dilkina, and Koenig, 2021), theorem proving (Balunovic, Bielik, and Vechev, 2018), evolutionary algorithms (Paliwal et al., 2020) and Bayesian optimization (J. Song, Y. Chen, and Yue, 2019).

Some research has taken a more high level view of the search process and aims to design end-to-end learning models to replace existing search framework. For example, several works on solving the traveling salesman problems (H. Dai, E. B. Khalil, et al., 2017; Kool, Hoof, and Welling, 2018; Joshi, Laurent, and Bresson, 2019) learn models that directly construct solutions from a problem description. Such models can also be used within a standard look-ahead search framework such as beam search or Monte-Carlo tree search to further boost their performances.

Algorithm Configuration Another area of using learning to speed up optimization solvers is algorithm configuration (Hoos, 2011; Hutter, Hoos, and Leyton-Brown, 2011; Ansótegui et al., 2015; Balcan et al., 2018; Kleinberg et al., 2019). Existing solvers tend to have many customizable hyperparameters whose values strongly influence the solver behaviors. Algorithm configuration aims to optimize those parameters on a problem-by-problem basis to speed up the solver.

Algorithm portfolio (or algorithm selection) (Gomes and Selman, 2001) is another related area. If we think algorithm configuration as exploiting the performance variance as a consequence of solver parameters, algorithm portfolio exploits the variance among different solvers. This class of methods learn a selection model that dispatches different solvers based on problem instance features. Though conceptually simple, they have shown outstanding performance in applications from SAT solving (Xu et al., 2008) to numerical optimization (Peng et al., 2010).

Learning to Identify Substructures The third category of approaches is learning to predict key substructures of an optimization problem. A canonical example is learning to predict backdoor variables (R. Williams, Gomes, and Selman, 2003;

Dilkina, Gomes, and Sabharwal, 2009), which are a set of variables that, once instantiated, the remaining problem simplifies to a tractable form (Dilkina, Gomes, and Sabharwal, 2009). Kilby et al. (2005) propose algorithms for finding backdoors for SAT by collecting branching variables from a SAT solver. For MIP solving, Fischetti and Monaci (2011) design a method to identify pseudo-backdoors for a given MIP by solving a set covering problem, and observe that by branching on variables in a pseudo-backdoor first a MIP solver can solve practical instances from MIPLIB (Koch et al., 2011) faster. Dvořák et al. (2017) study a variant of backdoors for MIPs called fracture backdoors which are variables whose removal would result in a natural decomposition of MIPs. Our method is the first data-driven attempt to predict pseudo-backdoors directly from MIP instances.

Other examples of this general paradigm include learning to pre-condition solvers, such as generating an initial solution to be refined with a downstream solver, which is typically more popular in continuous optimization settings (Kim et al., 2018).

End-to-end Differentiable Optimization The final category views optimizing an optimization algorithm as a meta-problem. A common approach is to define some parametrization of an algorithm and formulate a *differentiable* objective. The work by (Andrychowicz et al., 2016) uses LSTMs to predict the learning rate for gradient descent algorithms. The recurrent nature of the algorithm makes the whole pipeline differentiable. While it is easier to tackle continuous optimization problems with this perspective, existing research has shown that clever algorithm designs can extend the applicability into discrete optimization as well. Vlastelica et al. (2019) and Rolínek et al. (2020) present algorithms to differentiate through blackbox combinatorial solvers.

Another fascinating line of research studies embedding optimization algorithms as differentiable layers into other models. Amos and Kolter (2017) embeds quadratic programs into deep neural networks. Recent works (P.-W. Wang et al., 2019; Ferber, Wilder, et al., 2020; Wilder et al., 2019) explore applications in SAT solving, MIP solving and graph clustering.

2.2 Policy Learning for Sequential Decision-Making

Sequential decision making pertains to tasks where the policy performs a series of actions in a stateful environment. A popular framework to characterize the interaction between the agent and the environment is a Markov Decision Process

(MDP). There are two main approaches for policy learning in MDPs: reinforcement learning and imitation learning.

Reinforcement learning Reinforcement learning (RL) uses the observed environmental rewards to perform policy optimization. Recent works include Q-Learning approaches such as deep Q-networks (Mnih, Kavukcuoglu, Silver, Graves, et al., 2013), as well as policy gradient approaches such as DDPG (Lillicrap et al., 2015), TRPO (Schulman, Levine, et al., 2015) and PPO (Schulman, Wolski, et al., 2017). Despite its successful applications to a wide variety of tasks including playing games (Mnih, Kavukcuoglu, Silver, Graves, et al., 2013; Silver et al., 2016), robotics (Levine et al., 2016; Kober, J. A. Bagnell, and Peters, 2013) and combinatorial optimization (H. Dai, E. B. Khalil, et al., 2017; Mirhoseini et al., 2017), high sample complexity and unstable learning pose significant challenges in practice (Henderson et al., 2018), often causing learning to be unreliable.

Imitation Learning Imitation learning (IL) uses demonstrations (from an expert) as the primary learning signal. One popular class of algorithms is reduction-based (Daumé III, Langford, and Marcu, 2009; Stéphane Ross and D. Bagnell, 2010; Stéphane Ross, Gordon, and D. Bagnell, 2011; Stephane Ross and J. A. Bagnell, 2014; Chang et al., 2015), which generates cost-sensitive supervised examples from demonstrations. Other approaches include estimating the expert's cost-to go (W. Sun et al., 2017), inverse reinforcement learning Abbeel and Ng, 2004; Ho and Ermon, 2016; Ziebart et al., 2008, and behavioral cloning U. Syed and Schapire, 2008. One major limitation of imitation learning is the reliance on demonstrations. One solution is to combine imitation and reinforcement learning (H. Le et al., 2018; Kang, Jie, and Feng, 2018; Cheng et al., 2018; Nair et al., 2018) to learn from fewer or coarser demonstrations.

Chapter 3

LEARNING TO SEARCH WITH RETROSPECTIVE IMITATION

Abstract We study the problem of learning a good search policy for combinatorial search spaces. We propose retrospective imitation learning, which, after initial training by an expert, improves itself by learning from *retrospective inspections* of its own roll-outs. That is, when the policy eventually reaches a feasible solution in a combinatorial search tree after making mistakes and backtracks, it retrospectively constructs an improved search trace to the solution by removing backtracks, which is then used to further train the policy. A key feature of our approach is that it can iteratively scale up, or transfer, to larger problem sizes than those solved by the initial expert demonstrations, thus dramatically expanding its applicability beyond that of conventional imitation learning. We showcase the effectiveness of our approach on a range of tasks, including synthetic maze solving and combinatorial problems expressed as integer programs.

3.1 Introduction

Many challenging tasks involve traversing a combinatorial search space. Examples include branch-and-bound for constrained optimization problems (Lawler and Wood, 1966), A* search for path planning (Hart, Nilsson, and Raphael, 1968) and game playing, e.g., Go (Silver et al., 2016). Since the search space often grows exponentially with problem size, one key challenge is how to prioritize traversing the search space. A conventional approach is to manually design heuristics that exploit specific structural assumptions (cf. Gonen and Lehmann (2000) and Holmberg and Yuan (2000)). However, this conventional approach is labor intensive and relies on human experts developing a strong understanding of the structural properties of some class of problems.

In this paper, we take a learning approach to finding an effective search heuristic. We cast the problem as policy learning for sequential decision making, where the environment is the combinatorial search problem. Viewed in this way, a seemingly natural approach to consider is reinforcement learning, where the reward comes from finding a feasible terminal state, e.g., reaching the target in A* search. However, in our problem, most terminal states are not feasible, so the reward signal is sparse; hence, we do not expect reinforcement learning approaches to be effective.

We instead build upon imitation learning (Stéphane Ross and D. Bagnell, 2010; Stéphane Ross, Gordon, and D. Bagnell, 2011; Daumé III, Langford, and Marcu, 2009; He, Daume III, and Eisner, 2014), which is a promising paradigm here since an initial set of solved instances (i.e., demonstrations) can often be obtained from existing solvers, which we also call experts. However, obtaining solved instances can be expensive, especially for large problems. Hence, one key challenge is to avoid repeatedly querying experts during *training*.

We propose the *retrospective imitation* approach, where the policy can iteratively learn from its own mistakes without repeated expert feedback. Instead, we use a *retrospective oracle* to generate feedback by querying the environment on rolled-out search traces (e.g., which part of the trace led to a feasible terminal state) to find the shortest path in hindsight (retrospective optimal trace).

Our approach improves upon previous imitation approaches (Stéphane Ross and D. Bagnell, 2010; Stéphane Ross, Gordon, and D. Bagnell, 2011; He, Daume III, and Eisner, 2014) in two aspects. First, our approach iteratively refines towards solutions that may be higher quality or easier for the policy to find than the original demonstrations. Second and more importantly, our approach can scale to larger problem instances than the original demonstrations, allowing our approach to scale up to problem sizes beyond those that are solvable by the expert, and dramatically extending the applicability beyond that of conventional imitation learning. We also provide a theoretical characterization for a restricted setting of the general learning problem.

We evaluate on two types of search problems: A* search and branch-and-bound in integer programs. We demonstrate that our approach improves upon prior imitation learning work (He, Daume III, and Eisner, 2014) as well as commercial solvers such as Gurobi (for integer programs). We further demonstrate generalization ability by learning to solve larger problem instances than contained in the original training data.

In summary, our contributions are:

- We propose retrospective imitation, a general learning framework that generates feedback (via querying the environment) for imitation learning, without repeatedly querying experts.
- We show how retrospective imitation can scale up beyond the problem size where demonstrations are available, which significantly expands upon the

capabilities of imitation learning.

- We provide theoretical insights on when retrospective imitation can provide improvements over imitation learning, such as when we can reliably scale up.

- We evaluate empirically on three combinatorial search environments and show improvements over both imitation learning baselines and off-the-shelf solvers.

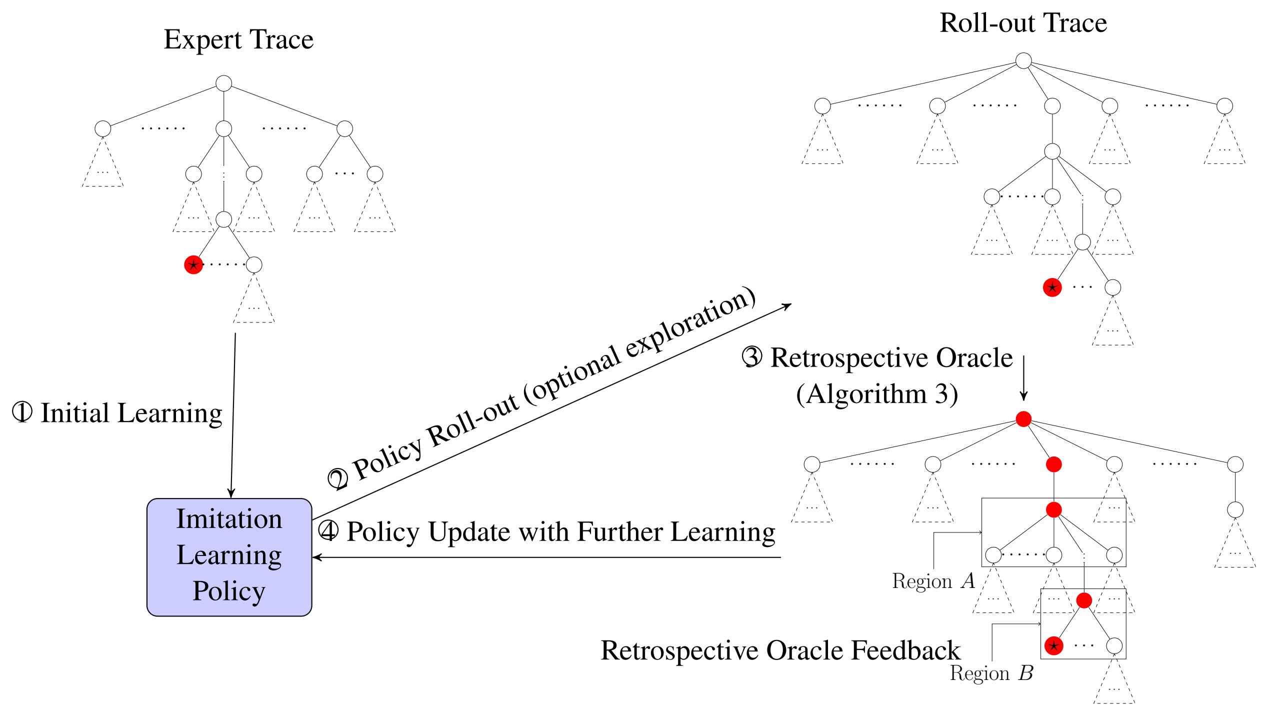
3.2 Problem Setting & Preliminaries

Learning Search Policies for Combinatorial Search Problems. Given a combinatorial search problem P, a policy π (i.e., a search algorithm) must make a sequence of decisions to traverse a combinatorial search space to find a (good) feasible solution (e.g., a setting of integer variables in an integer program that satisfies all constraints and has good objective value). We focus on combinatorial tree search, where the navigation of the search space is organized as search trees, i.e., our "environment" is the search space. Given the current "state" s_t (the current search tree), which contains the search history so far (e.g., a partial assignment of integer variables), the policy chooses an action a, usually a new node to explore, to apply to the current state s_t (i.e., to extend the current partial solution) and transitions to a new state s_{t+1} (a new search tree). The search terminates when a complete feasible solution is found, which we also refer to as reaching a terminal state. Figure 3.1 depicts (among other things) example roll-outs, or search traces, of such policies.

A typical objective is to minimize search time to a terminal state. In general, the transition function is deterministic and known, but navigating a combinatorial search space to find rare terminal states is challenging. Given a training set of problem instances, we can use a learning approach to train π to perform well on test problem instances.

Imitation Learning. We build upon the imitation learning paradigm to learn a good search policy. Previous work assumes an expert policy π_{expert} that provides interactive feedback on the trained policy He, Daume III, and Eisner, 2014. The expert can be a human or an (expensive) solver. However, repeated queries to the expert can be prohibitively expensive.

Our approach is based on the idea that retrospection (with query access to environment) can also generate feedback. A search trace typically has many dead ends and backtracking before finding a terminal state. Thus, more efficient search traces (i.e., feedback) can be retrospectively extracted by removing backtracking, which forms



Expert Trace
Roll-out Trace
① Initial Learning
② Policy Roll-out (optional exploration)
③ Retrospective Oracle (Algorithm 3)
Imitation Learning Policy
④ Policy Update with Further Learning
Region A
Retrospective Oracle Feedback
Region B

Figure 3.1: A visualization of retrospective imitation learning depicting components of Algorithm 1. An imitation learning policy is initialized from expert traces and is rolled out to generate its own traces. Then the policy is updated according to the feedback generated by the retrospective oracle as in Figure 3.2. This process is repeated until some termination condition is met.

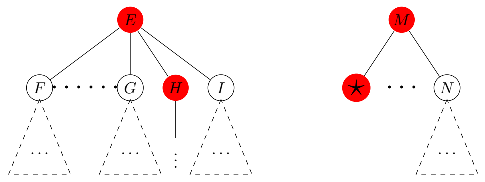


Figure 3.2: Zoom-in views of Region A and B in Figure 3.1. At node E, the retrospective feedback indicates selecting node H over F, G, and I. At node M, the $\star$ node is preferred over N.

the core algorithmic innovation of our approach (see Section 3.3). Retrospective imitation also enables a form of transfer learning, where we iteratively train policies to solve larger problems for which collecting demonstrations is infeasible (e.g., due to computational costs of the original expert solver).

3.3 Retrospective Imitation Learning

We now describe the retrospective imitation learning approach. It is a general framework that can be combined with a variety of imitation learning algorithms. We

Algorithm 1 Retrospective DAgger for Fixed Size

```
 1: Inputs:, $N$ the number of iterations, $\pi_1$ an initial policy trained on expert traces,
    $\alpha$ the mixing parameter, $\{P_j\}$ a set of training problem instances, $D_0$ the expert
    traces dataset
 2: initialize $D = D_0$
 3: for $i = 1, \cdots, N$ do
 4:     $\hat{\pi_i} \leftarrow \alpha\pi_i + (1-\alpha)\pi_{explore}$   (optionally explore)
 5:     run $\hat{\pi_i}$ on $\{P_j\}$ to generate a set of search traces $\{\tau_j\}$
 6:     for each $\tau_j$, compute $\pi^*(\tau_j, s)$ for each terminal state $s$ (Algorithm 3)
 7:     collect new dataset $D_i$ based on each $\pi^*(\tau_j, s)$
 8:     update $D$ with $D_i$ (i.e., $D \leftarrow D \cup D_i$)
 9:     train $\pi_{i+1}$ on $D$
10: end for
11: return best $\pi_i$ on validation
```

instantiate our approach using the data aggregation algorithm (DAgger) (Stéphane Ross, Gordon, and D. Bagnell, 2011; He, Daume III, and Eisner, 2014) and we call the resulting algorithm Retrospective DAgger (Algorithm 1). We also include the instantiation with SMILe (Stéphane Ross and D. Bagnell, 2010) and call it Retrospective SMILe (Algorithm 2). In Section 3.5, we empirically evaluate retrospective imitation with both DAgger and SMILe to showcase the generality of our framework. For clarity of presentation, our presentation will focus on Retrospective DAgger.

We decompose our general framework into two steps. First, Algorithm 1 describes our core procedure for learning on fixed size problems with a crucial *retrospective oracle* subroutine (Algorithm 3). Algorithm 4 then describes how to scale up beyond the fixed size. We will use Figure 3.1 as a running example. The ultimate goal is to enable imitation learning algorithms to scale up to problems much larger than those for which we have expert demonstrations, which is a significant improvement since conventional imitation learning cannot naturally accomplish this.

Core Algorithm for Fixed Problem Size. We assume access to an initial dataset of expert demonstrations to help bootstrap the learning process, as described in Line 3 in Algorithm 1 and depicted in step ① in Figure 3.1. Learning proceeds iteratively. In Lines 9-10, the current policy (potentially blended with an exploration policy) runs until a termination condition, such as reaching one or more terminal states, is met. In Figure 3.1, this is step ② and the red node is a terminal state. In Line 11, a retrospective oracle computes the retrospective optimal trace for each terminal state (step ③). This is identified by the path with red nodes from the root to the terminal state. In Line 12, a new dataset is generated, as discussed below. In Lines 12-14, we

Algorithm 2 Retrospective SMILe for Fixed Size

```
1: Inputs: $N$ number of iterations, $\pi_1$ an initial policy trained on expert traces, $\alpha$
   the mixing parameter, $\{P_j\}$ a set of problem instances
2: for $i = 1, \cdots, N$ do
3:     run $\pi_i$ on $\{P_j\}$ to generate trace $\{\tau_j\}$
4:     compute $\pi^*(\tau_j, s)$ for each terminal state $s$ (Algorithm 3)
5:     collect new dataset $D$ based on $\pi^*(\tau_j, s)$
6:     train $\hat{\pi}_{i+1}$ on $D$
7:     $\pi_{i+1} = (1-\alpha)^i \pi_1 + \alpha \sum_{j=1}^{i} (1-\alpha)^{j-1} \hat{\pi}_j$
8: end for
9: return $\pi_{N+1}$
```

Algorithm 3 Retrospective Oracle for Tree Search

```
1: Inputs: $\tau$ a search tree trace, $s \in \tau$ a terminal state
2: Output: retro_optimal: the retrospective optimal trace
3: while $s$ is not the root do
4:     parent ← $s$.parent
5:     retro_optimal(parent) ← $s$
6:     $s$ ← parent
7: end while
8: return retro_optimal
```

Algorithm 4 Retrospective Imitation for Scaling Up

```
1: Inputs: $S_1$ the initial problem size, $S_2$ the target problem size, $\pi_{S_1}$ a policy
   trained on expert data of problem size $S_1$
2: for $s = S_1 + 1, \cdots, S_2$ do
3:     generate problem instances $\{P_i^s\}$ of size $s$
4:     train $\pi_s$ via Alg. 1 by running $\pi_{s-1}$ on $\{P_i^s\}$ to generate initial search traces
5: end for
```

imitate the retrospective optimal trace (in this case using DAgger) to obtain a much more efficient search policy. We then train a new policy and repeat the process.

Retrospective Oracle. A retrospective oracle (with query access to the environment) takes as input a search trace τ and outputs a retrospective optimal trace $\pi^*(\tau, s)$ for each terminal state s. Note that optimality is measured with respect to τ, and not globally. That is, based on τ, what is the fastest/shortest known action sequence to reach a terminal state if we were to solve the *same* instance again? In Figure 3.1, given the current trace with a terminal state $\star$ (step ②), the retrospective optimal trace is the path along red nodes (step ③). In general, $\pi^*(\tau, s)$ will be shorter than τ, which implies faster search in the original problem. Algorithm 3 shows the retrospective oracle for tree-structured search. Identifying a retrospective optimal trace given a terminal state is equivalent to following parent pointers until the initial state, as this results in the shortest trace.

Design Decisions in Training Data Creation. Algorithm 1 requires specifying how to create each new dataset D_i given the search traces and the retrospective optimal ones (Line 12 of Algorithm 1). Intuitively D_i should show how to correct mistakes made during roll-out to reach a terminal state s. What constitutes a mistake is influenced by the policy's actions. For reduction-based imitation learning algorithms such as DAgger and SMILe, the learning reduction converts the retrospective optimal solution into per state-action level supervised learning labels. Two concrete examples are shown in Figure 3.2.

Furthermore, in the case that τ contains multiple terminal states, we also need to decide which to prioritize. See Section 3.5 for concrete instantiations of these decisions for learning search heuristics for solving mazes and learning branch-and-bound heuristics for solving integer programs.

Scaling Up. The most significant benefit of retrospective imitation is the ability to scale up to problems of sizes beyond those in the initial dataset of expert demonstrations. Algorithm 4 describes our overall framework, which iteratively learns to solve increasingly larger instances using Algorithm 1 as a subroutine. We show in the theoretical analysis that, under certain assumptions, retrospective imitation is guaranteed able to scale, or transfer, to increasingly larger problem instances. The basic intuition is that slightly larger problem instances are often "similar enough" to the current size problem instances, so that the current learned policy can be used as the initial expert when scaling up.

Incorporating Exploration. In practice, it can be beneficial to employ some exploration. Exploration is typically more useful when scaling up to larger problem instances. In our experiments, we have found the following two strategies to be most useful.

- ϵ-greedy strategy allows a certain degree of random exploration. This helps learned policies to discover new terminal states and enables retrospective imitation learning to learn from a more diverse goal set. Discovering new terminal states is especially important when scaling up because the learned policies are trained for a smaller problem size; to counter the domain shift when scaling up, we add exploration to enable the learned policies to find better solutions for the new larger problem size.

- Searching for multiple terminal states and choosing the best one as the learning target. This is an extension to the previous point since by comparing multiple terminal states, we can pick out the one that is best for the policy to target, thus improving the efficiency of learning.

- When scaling up, for the first training pass on each problem scale, we collect multiple traces on each data point by injecting 0.05 variance Gaussian noise into the regression model within the policy class, before choosing the best feasible solution.

3.4 Theoretical Results

Summary of Theoretical Results

In this section, we provide theoretical insights on when we expect retrospective imitation to improve reduction based imitation learning algorithms, such as DAgger and SMILe.

For simplicity, we regard all terminal states as equally good, so we simply aim to find one as quickly as possible. Note that our experiments evaluate settings beyond those covered in the theoretical analysis.

Our analysis builds on a **trace inclusion assumption**: *the search trace τ_1 generated by a trained policy contains the trace τ_2 by an expert policy.* While somewhat strict, this assumption allows us to rigorously characterize retrospective imitation when scaling up. We measure the quality of a policy using the following error rate:

$$\epsilon = \frac{\#\text{Non-optimal actions compared to retrospective optimal trace}}{\#\text{Actions to reach a terminal state in retrospective optimal trace}}.$$

Intuitively, this metric measures how often a policy fails to agree with the retrospective oracle. The following proposition states that retrospective imitation can effectively scale up and obtain a lower error rate.

Proposition 1. *Let π_{S_1} be a policy trained using imitation learning on problem size S_1. If, during the scaling-up training process to problems of size $S_2 > S_1$, the trained policy search trace, starting from π_{S_1}, always contains the (hypothetical) expert search trace on problem of size S_2 (trace inclusion assumption), then the final* *error rate ϵ_{S_2} is at most that obtained by running imitation learning (with expert demonstrations) directly on problems of size S_2.*

Proof. By the trace inclusion assumption, the dataset obtained by retrospective imitation will contain feedback for every node in the expert trace. Furthermore, the retrospective oracle feedback corresponds to the right training objective while the dataset collected by imitation learning does not, as explained in Section 3.3. So the error rate trained on retrospective imitation learning data will be at most that of imitation learning. $\square$

Next we analyze how lower error rates impact the number of actions to reach a terminal state. We restrict ourselves to decision spaces of size 2: branch to one of its children or backtrack to its parent. Theorem 2 equates the number of actions to hitting time for an asymmetric random walk.

Theorem 2. *Let π be a trained policy that has an error rate of $\epsilon \in (0, \frac{1}{2})$ as measured against the retrospective feedback. Let P be a search problem where the optimal action sequence has length N, and let T be the number of actions by π to reach a terminal state. Then the expected number of actions by π to reach a terminal state is $\mathbb{E}[T] = \frac{N}{1-2\epsilon}$. Moreover, $\mathbb{P}[T \geq \alpha N] \in O(\exp(-\alpha + \mathbb{E}[T]/N))$ for any $\alpha \geq 0$.*

This result implies that lower error rates lead to shorter search time (in the original search problem) with exponentially high probability. By combining this result with the lower error rate of retrospective imitation (Proposition 1), we see that retrospective imitation has a shorter expected search time than the corresponding imitation learning algorithm. We provide further analysis in the appendix.

Additional Proofs

To prove Theorem 2 we need the following lemma on asymmetric 1-dimensional random walks.

Lemma 3. *Let $Z_i, i = 1, 2, \cdots$ be i.i.d. Bernoulli random variables with the distribution*

$$Z_i = \begin{cases} 1, & \textit{with probability } 1-\epsilon \\ -1, & \textit{with probability } \epsilon \end{cases}$$

for some $\epsilon \in [0, \frac{1}{2})$*. Define* $X_n = \sum_{i=1}^{n} Z_i$ *and the hitting time* $T_N = \inf\{n : X_n = N\}$ *for some fixed integer* $N \geq 0$*. Then* $\mathbb{E}[T_N] = \frac{N}{1-2\epsilon}$ *and* $\mathbb{P}[T_N \geq \alpha N] \in O(\exp(-\alpha + \mathbb{E}[T_N]/N))$*.*

Proof. The proof will proceed as follows: we will begin by computing the moment-generating function (MGF) for T_1 and then use it to compute the MGF of T_N. Then, we will use this MGF to produce a Chernoff-style bound on $\mathbb{P}[T_N > \alpha N]$.

The key observation for computing the MGF of T_N is that $T_N \overset{\text{dist}}{=} \sum_{i=1}^{N} T_1^{(i)}$, where $T_1^{(1)}, \ldots, T_1^{(N)} \overset{\text{iid}}{\sim} \mathbb{P}[T_1]$. This is because the random walk moves by at most one position at any given step, independent of its overall history. Therefore the time it takes the walk to move N steps to the right is exactly the time it takes for the walk to move 1 step to the right N times. So we have $\mathbb{E}\left[e^{\beta T_N}\right] = \mathbb{E}\left[e^{\beta T_1}\right]^N$ by independence.

With this in mind, let $\Phi(\lambda) = \mathbb{E}\left[\lambda^{T_1}\right]$ be the generating function of T_1. Then, by the law of total expectation and the facts above,

$$\begin{aligned}
\Phi(\lambda) &= \mathbb{P}[Z_1 = 1]\,\mathbb{E}\left[\lambda^{T_1} \mid Z_1 = 1\right] \\
&\quad + \mathbb{P}[Z_1 = -1]\,\mathbb{E}\left[\lambda^{T_1} \mid Z_1 = -1\right] \\
&= (1-\epsilon)\mathbb{E}\left[\lambda^{1+T_0}\right] + \epsilon\mathbb{E}\left[\lambda^{1+T_2}\right] \\
&= \lambda\left((1-\epsilon)\mathbb{E}\left[\lambda^{1+T_0}\right] + \epsilon\mathbb{E}\left[\lambda^{1+T_1^{(1)}+T_1^{(2)}}\right]\right) \\
&= \lambda\left((1-\epsilon) + \epsilon\Phi(\lambda)^2\right).
\end{aligned}$$

Solving this quadratic equation in $\Phi(\lambda)$ and taking the solution that gives $\Phi(0) = \mathbb{E}\left[0^{T_1}\right] = 0$, we get

$$\Phi(\lambda) = \frac{1}{2\epsilon^2\lambda}\left(1 - \sqrt{1 - 4\epsilon(1-\epsilon)\lambda^2}\right).$$

Now, we note that the MGF of T_1 is just $\Phi(e^\beta) = \mathbb{E}\left[e^{\beta T_1}\right]$. So $\mathbb{E}\left[e^{\beta T_N}\right] = \Phi(e^\beta)^N$. To prove the first claim, we can just differentiate $\mathbb{E}\left[e^{\beta T_N}\right]$ in β and evaluate it at $\beta = 0$, which tells us that $\mathbb{E}[T_N] = \frac{N}{1-2\epsilon}$.

To prove the second claim, we can apply Markov's inequality to conclude that for any $\alpha, \beta \geq 0$,

$$\begin{aligned}
\mathbb{P}[T_N \geq \alpha N] &= \mathbb{P}\left[e^{\beta T_N} \geq e^{\beta\alpha N}\right] \\
&\leq \mathbb{E}\left[e^{\beta T_N} e^{-\beta\alpha N}\right] \\
&= \left(\mathbb{E}\left[e^{\beta T_1}\right] e^{-\beta\alpha}\right)^N.
\end{aligned}$$

Letting $\beta = \frac{1}{N}$ and taking the limit as $N \to \infty$, we get that

$$\lim_{N\to\infty} \mathbb{P}\left[T_N \geq \alpha N\right] \leq \exp\left(-\alpha + \frac{1}{1-2\epsilon}\right).$$

which implies the concentration bound asymptotically. $\square$

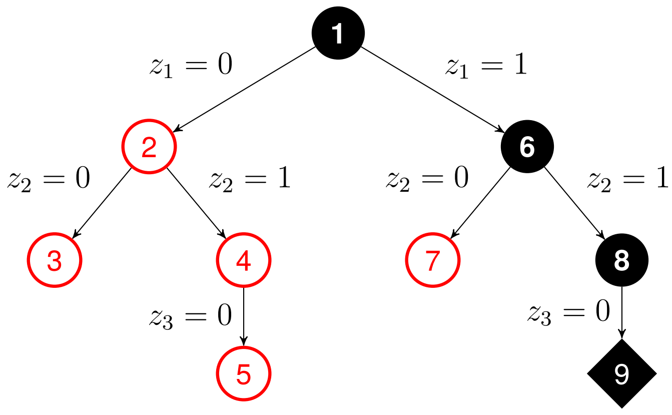


Figure 3.3: An example search trace by a policy. The solid black nodes ($1 \to 6 \to 8 \to 9$) make up the best trace to a terminal state in retrospect. The empty red nodes are the mistakes made during this search procedure. Every mistake increases the distance to the target node (node 9) by 1 unit, while every correct decision decreases the distance by 1 unit.

Now onto the proof for the Theorem 2.

Proof. We consider the search problem as a 1-dimensional random walk (see Figure 3.3). The random walk starts at the origin and proceeds in an episodic manner. The goal is to reach the point N and at each time step, a wrong decision is equivalent to moving 1 unit to the left whereas a right decision is equivalent to moving 1 unit to the right. The error rate of the policy determines the probabilities of moving left and right. Thus the search problem can be reduced to 1-dimensional random walk, so we can invoke the previous lemma and assert (1) that the expected number of time steps before reaching a feasible solution is $\frac{N}{1-2\epsilon}$, and (2) that the probability that this number of time steps is greater than αN is $O\left(\exp\left(-\alpha + \frac{1}{1-2\epsilon}\right)\right)$. $\square$

This theorem allows us to measure the impact of error rates on the expected number of actions.

Corollary 3.1. *With two policies π_1 and π_2 with corresponding error rates $0 < \epsilon_1 < \epsilon_2 < \frac{1}{2}$, π_2 takes $\frac{1-2\epsilon_1}{1-2\epsilon_2}$ times more actions to reach a feasible state in expectation.*

Moreover, the probability that π_1 terminates in αN time steps (for any $\alpha \geq 0$) is $\exp\left(\frac{1}{1-2\epsilon_2} - \frac{1}{1-2\epsilon_1}\right)$ times higher.

3.5 Experimental Results

We empirically validate the generality of our retrospective imitation technique by instantiating it with two well-known imitation learning algorithms, DAgger (Stéphane Ross, Gordon, and D. Bagnell, 2011) and SMILe (Stéphane Ross and D. Bagnell, 2010). We showcase the scaling up ability of retrospective imitation by only using demonstrations on the smallest problem size and scaling up to larger sizes in an entirely unsupervised fashion through Algorithm 3. We experimented on both A* search and branch-and-bound search for integer programs.

Environments and Datasets

We experimented on three sets of tasks, as described below.

Maze Solving with A* Search. We generate random mazes according to the Kruskal's algorithm (Kruskal, 1956). For imitation learning, we use search traces provided by an A* search procedure equipped with the Manhattan distance heuristic as initial expert demonstrations.

We experiment on mazes of 5 increasing sizes, from 11×11 to 31×31. For each size, we use 48 randomly generated mazes for training, 2 for validation and 100 for testing. We perform A* search with Manhattan distance as the search heuristic to generate initial expert traces which are used to train imitation learning policies. The learning task is to learn a priority function to decide which locations to prioritize and show that it leads to more efficient maze solving. For our retrospective imitation policies, we only assume access to expert traces of maze size 11×11 and learning on subsequent sizes is carried out according to Algorithm 3. Running retrospective imitation resulted in generating $\sim$ 100k individual data points.

Integer Programming for Risk-aware Path Planning. We consider the risk-aware path planning problem from Ono, B. C. Williams, and Blackmore (2013). Our formulation is based on the MILP-based path planning originally presented by (Schouwenaars et al., 2001), combined with risk-bounded constrained tightening (Prékopa, 1999). It is a similar formulation as that of the state-of-the-art risk-aware path planner pSulu (Ono, B. C. Williams, and Blackmore, 2013) but without risk allocation.

We consider a path planning problem in $\mathbb{R}^{\Bbbk}$, where a path is represented as a sequence of N way points $x_1, \cdots x_N \in X$. The vehicle is governed by a linear dynamics given by:

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + w_k \\ u_k &\in U, \end{aligned}$$

where $U \subset \mathbb{R}^m$ is a control space, $u_k \in U$ is a control input, $w_k \in \mathbb{R}^n$ is a zero-mean Gaussian-distributed disturbance, and A and B are n-by-n and n-by-m matices, respectively. Note that the dynamic of the mean and covariance of x_i, denoted by $\bar{x}_i$ and Σ_i, respectively, have a deterministic dynamics:

$$\begin{aligned} \bar{x}_{k+1} &= A\bar{x}_k + Bu_k + w_k \\ \Sigma_{k+1} &= A\Sigma A^T + W, \end{aligned} \tag{3.1}$$

where W is the covariance of w_k. We assume there are M polygonal obstacles in the state space, hence the following linear constraints must be satisfied in order to be safe (as in Figure 3.4):

$$\bigwedge_{k=1}^{N} \bigwedge_{i=1}^{M} \bigvee_{j=1}^{L_i} h_{ij} x_k \leq g_{ij},$$

where $\bigwedge$ is conjunction (i.e., AND), $\bigvee$ is disjunction (i.e., OR), L_i is the number of edges of the i-th obstacle, and h_{ij} and g_{ij} are constant vector and scaler, respectively. In order for each of the linear constraints to be satisfied with the probability of $1-\delta_{kij}$, the following has to be satisfied:

$$\begin{aligned} &\bigwedge_{k=1}^{N} \bigwedge_{i=1}^{M} \bigvee_{j=1}^{L_i} h_{ij} \bar{x}_k \leq g_{ij} - \Phi(\delta_{kij}) \\ &\Phi(\delta_{kij}) = -\sqrt{2 h_{ijk} \Sigma_{x,k} h_{ijk}^T} \, \mathrm{erf}^{-1}(2\delta_{ijk} - 1), \end{aligned} \tag{3.2}$$

where erf^{-1} is the inverse error function.

The problem that we solve is, given the initial state $(\bar{x}_0, \Sigma_0)$, to find $u_1 \cdots u_N \in U$ that minimizes a linear objective function and satisfies (3.1) and (3.2). An arbitrary nonlinear objective function can be approximated by a piecewise linear function by introducing integer variables. The disjunction in (3.2) is also replaced by integer variables using the standard Big M method. Therefore, this problem is equivalent to MILP. In the branch-and-bound algorithm, the choice of which linear constraint to be satisfied among the disjunctive constraints in (3.2) (i.e., which side of the obstacle x_k is) corresponds to which branch to choose at each node.

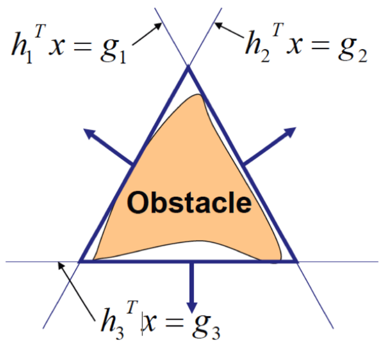


Figure 3.4: Representation of polygonal obstacle by disjuctive linear constraints.

We generate 150 obstacle maps. Each map contains 10 rectangle obstacles, with the center of each obstacle chosen from a uniform random distribution over the space $0 \leq y \leq 1$, $0 \leq x \leq 1$. The side length of each obstacle was chosen from a uniform distribution in range $[0.01, 0.02]$ and the orientation was chosen from a uniform distribution between $0°$ and $360°$. In order to avoid trivial infeasible maps, any obstacles centered close to the destination are removed. The risk bound was set to $\delta = 0.02$. We started from problems with 10 way points and scaled up to 14 way points, in increments of 1. The number of integer variables range from 400 to 560, which can be quite challenging to solve. For training, we assume that expert demonstrations by Gurobi are only available for the smallest problem size (10 way points, 400 binary variables). We use 50 instances for each of training, validation and testing. Running retrospective imitation resulted in generating ~ 1.4 million individual data points.

Integer Programming for Minimum Vertex Cover. Minimum vertex cover (MVC) is a classical NP-hard combinatorial optimization problem, where the goal is to find the smallest subset of nodes in a given graph, such that every edge is adjacent to at least one node in this subset. This problem is quite challenging, and is difficult for commercial solvers even with large computational budgets. We generate random Erdős-Renyi graphs (Erdős and Rényi, 1960) with varying number of nodes from 100 to 500. For each graph, its MVC problem is compiled into an integer linear program (ILP) and we use also the branch-and-bound search method to solve it. The number of integer variables range from 100 to 500. We use 15 labeled and 45 unlabeled graphs for training and test on 100 new graphs for each scale. Running retrospective imitation resulted in generating ~ 350k individual data points.

Policy Learning

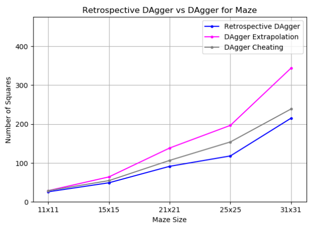
For A* search, we learn a ranking model as the policy. The input features are mazes represented as a discrete-valued matrix indicating walls, passable squares, and the current location. We instantiate using neural networks with 2 convolutional layers with 32 3×3 filters each, 2×2 max pooling, and a feed-forward layer with 64 hidden units.

For branch-and-bound search in integer programs, we considered two policy classes. The first follows (He, Daume III, and Eisner, 2014), and consists of a node selection model (that prioritizes which node to consider next) and a pruning model (that rejects nodes from being branched on), which mirrors the structure of common branch-and-bound search heuristics. We use RankNet (Burges, Shaked, et al., 2005) as the selection model, instantiated using two layers with LeakyReLU (Maas, Hannun, and Ng, 2013) activation functions, and trained via cross entropy loss. For the pruning model, we train a 1-layer neural network classifier with higher cost on the optimal nodes compared to the negative nodes. We refer to this policy class as "select & pruner". The other policy class only has the node selection model and is referred to as "select only".

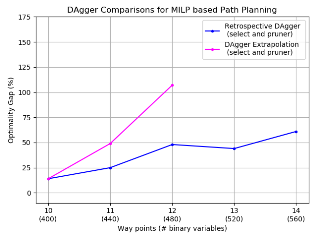
The features used can be categorized into node-specific and tree-specific features. Node-specific features include an LP relaxation lower bound, objective value and node depth. Tree-specific features capture global aspects that include the integrality gap, number of solutions found, and global lower and upper bounds. We normalize each feature to [-1,1] at each node, which is also known as query-based normalization (Qin, Jun, and Hang, 2010).

Main Results

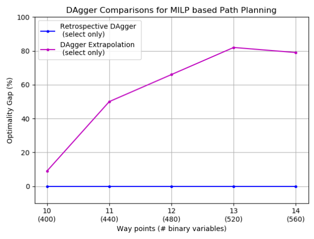
Comparing Retrospective Imitation with Imitation Learning. As retrospective learning is a general framework, we validate with two different baseline imitation learning algorithms, DAgger (Stéphane Ross, Gordon, and D. Bagnell, 2011) and SMILe (Stéphane Ross and D. Bagnell, 2010). We consider two possible settings for each baseline imitation learning algorithm. The first is “Extrapolation”, which is obtained by training an imitation model only using demonstrations on the smallest problem size and applying it directly to subsequent sizes without further learning. Extrapolation is the natural baseline to compare with retrospective imitation as both have access to the same demonstration dataset. The second baseline setting is “Cheating”, where we provide the baseline imitation learning algorithm with expert



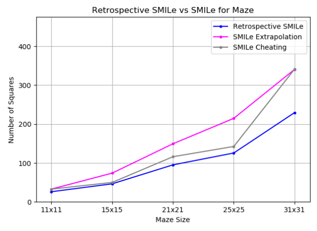
(a)



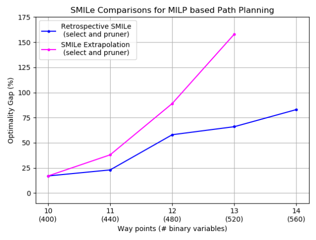
(b)



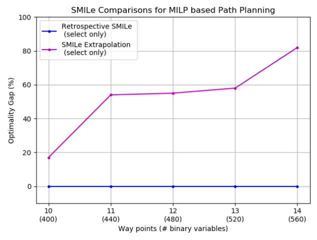
(c)



(d)



(e)



(f)

Figure 3.5: Retrospective imitation versus DAgger (top) and SMILe (bottom) for maze solving (left) and risk-aware path planning (middle and right). "Extrapolation" is the conventional imitation learning baseline, and "Cheating" (left column only) gives imitation learning extra training data. Retrospective imitation consistently and significantly outperforms imitation learning approaches in all settings.

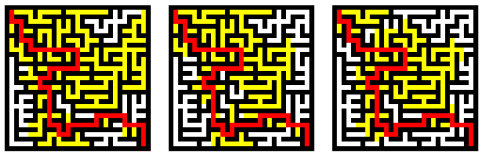
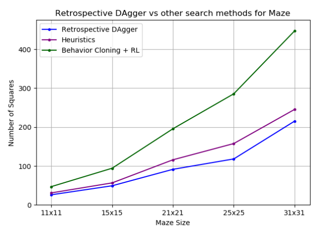
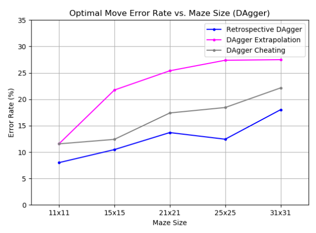


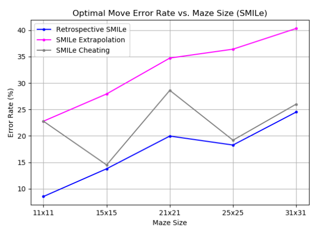
Figure 3.6: Left to right: comparing Manhattan distance heuristic, DAgger Cheating and Retrospective DAgger on a 31×31 maze starting at upper left and ending at lower right. Yellow squares are explored. Optimal path is red. The three algorithms explore 333, 271 and 252 squares, respectively.

demonstrations on the target problem size, which is significantly more than provided to retrospective imitation. Note that Cheating is not feasible in practice for settings of interest.

Our main comparison results are shown in Figure 3.5. For this comparison, we focus on maze solving and risk-aware path planning, as evaluating the true optimality gap for minimum vertex cover is intractable. We see that retrospective imitation (blue) consistently and dramatically outperforms conventional Extrapolation imitation learning (magenta) in every setting. We see in Figure 3.5a, 3.5d that retrospective imitation even outperforms Cheating imitation learning, despite having only expert demonstrations on the smallest problem size. We also note that

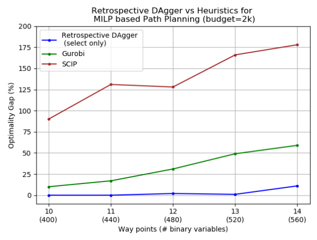


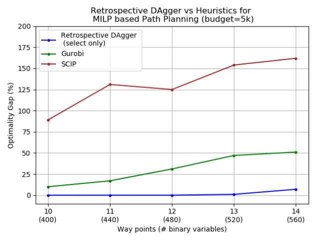


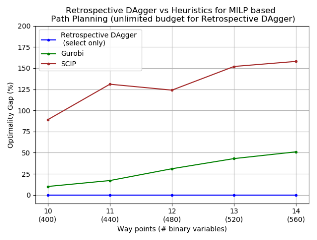


(a) (b) (c)

Figure 3.7: (left) Retrospective imitation versus off-the-shelf methods. The RL baseline performs very poorly due to sparse environmental rewards. (middle, right) Single-step decision error rates, used for empirically validating theoretical claims.







(a) (b) (c)

Figure 3.8: Retrospective DAgger ("select only" policy class) with off-the-shelf branch-and-bound solvers using various search node budgets. Retrospective DAgger consistently outperforms baselines.

Retrospective DAgger consistently outperforms Retrospective SMILe.

In the maze setting (Figure 3.5a, 3.5d), the objective is to minimize the number of explored squares to reach the target location. Without further learning beyond the base size, Extrapolation degrades rapidly and the performance difference with retrospective imitation becomes very significant. Even compared with Cheating policies, retrospective imitation still achieves better objective values at every problem size, which demonstrates its transfer learning capability. Figure 3.6 depicts a visual comparison for an example maze.

In the risk-aware path planning setting (Figure 3.5b, 3.5c, 3.5e, 3.5f), the objective is to find feasible solutions with low optimality gap, defined as the percentage difference between the best objective value found and the optimal (found via exhaustive search). If a policy fails to find a feasible solution we impose an optimality gap of 300% to arrive at a single comparison metric. We compare the optimality gap of the algorithms at the same number of explored nodes. In Figure 3.5b, 3.5e we first run the retrospective imitation version until termination, and then run the other

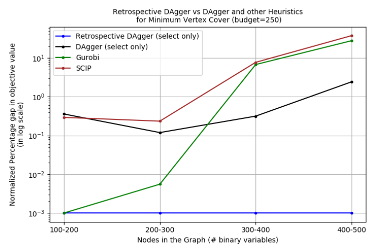


Figure 3.9: Relative objective value gaps of various methods compared with retrospective imitation when restricted with a search budget of 250 nodes. Retrospective imitation consistently outperforms other methods, especially at large scales.

algorithms to the same number of explored nodes. In Figure 3.5c, 3.5f, we first run the retrospective imitation with the "select only" policy class until termination, and then run the other algorithms to the same number of explored nodes. We note that the "select only" policy class (Figure 3.5c, 3.5f) significantly outperforms the "select and pruner" policy class (Figure 3.5b, 3.5e), which suggests that utilizing conceptually simpler policy classes may be more amenable to learning-based approaches in combinatorial search problems.

While scaling up, retrospective imitation obtains consistently low optimality gaps. In contrast, DAgger Extrapolation in Figure 3.5b failed to find feasible solutions for $\sim 60\%$ test instances beyond 12 way points, so we did not test it beyond 12 way points. SMILe Extrapolation in Figure 3.5e failed for $\sim 75\%$ of the test instance beyond 13 way points. The fact that retrospective imitation continues to solve larger MILPs with a very slow optimality gap growth suggests that our approach is performing effective transfer learning.

Minimum vertex cover is a challenging setting where it is infeasible to compute the optimal solution (even with large computational budgets). We thus plot relative differences in objective with respect to retrospective imitation. We see in Figure 3.9 that retrospective imitation consistently outperforms conventional imitation learning.

Comparing Retrospective Imitation with Off-the-Shelf Approaches. For maze solving, we compare with: 1) A* search with the Manhattan distance heuristic, and 2) behavioral cloning followed by reinforcement learning with a deep Q-network (Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015). Figure 3.7a shows Retrospective DAgger outperforming both methods. Due to the sparsity of the environmental rewards (only positive reward at terminal state), reinforcement learning performs significantly worse than even the Manhattan heuristic.

For risk-aware path planning and minimum vertex cover, we compare with a commercial solver Gurobi (Version 6.5.1) and SCIP (Version 4.0.1, using Gurobi as the LP solver). We implement our approach within the SCIP (Achterberg, 2009) integer programming framework. Due to differences in implementation, we use the number of explored nodes as a proxy for runtime. We control the search size for Retrospective DAgger ("select only") and use its resulting search sizes to control Gurobi and SCIP. Figures 3.8 & 3.9 show the results on a range of search size limits. We see that Retrospective DAgger ("select only") is able to consistently achieve the lowest optimality gaps, and the optimality gap grows very slowly as the number of integer variables scale far beyond the base problem scale. As a point of comparison, the next closest solver, Gurobi, has an optimality gap $\sim 50\%$ higher than Retrospective DAgger ("select only") at 14 waypoints (560 binary variables) in the risk-aware path planning task and a performance gap of $\sim 40\%$ compared with Retrospective DAgger at the largest graph scale for minimum vertex cover.

Empirically Validating Theoretical Results. Finally, we evaluate how well our theoretical results in Section 3.4 characterizes our experimental results. Figure 3.7b and 3.7c presents the optimal move error rates for the maze experiment, which validates Proposition 1 that retrospective imitation is guaranteed to result in a policy that has lower error rates than imitation learning. The benefit of having a lower error rate is explained by Theorem 2, which informally states that a lower error rate leads to shorter search time. This result is also verified by Figure 3.5a and 3.5d, where Retrospective DAgger/SMILe, having the lowest error rates, explores the smallest number of squares at each problem scale.

Chapter 4

CO-TRAINING FOR POLICY LEARNING

Abstract We study the problem of learning sequential decision-making policies in settings with multiple state-action representations. Such settings naturally arise in many domains, such as planning (e.g., multiple integer programming formulations) and various combinatorial optimization problems (e.g., those with both integer programming and graph-based formulations). Inspired by the classical co-training framework for classification, we study the problem of co-training for policy learning. We present sufficient conditions under which learning from two views can improve upon learning from a single view alone. Motivated by these theoretical insights, we present a meta-algorithm for co-training for sequential decision making. Our framework is compatible with both reinforcement learning and imitation learning. We validate the effectiveness of our approach across a wide range of tasks, including discrete/continuous control and combinatorial optimization.

4.1 Introduction

Conventional wisdom in problem-solving suggests that there is more than one way to look at a problem. For sequential decision making problems, such as those in reinforcement learning and imitation learning, one can often utilize multiple different state-action representations to characterize the same problem. A canonical application example is learning solvers for hard optimization problems such as combinatorial optimization (He, Daume III, and Eisner, 2014; Mirhoseini et al., 2017; H. Dai, E. B. Khalil, et al., 2017; J. Song, Lanka, Zhao, et al., 2018; Balunovic, Bielik, and Vechev, 2018). It is well-known in the operations research community that many combinatorial optimization problems have multiple formulations. For example, the maximum cut problem admits a quadratic integer program as well as a linear integer program formulation (Boros and Hammer, 1991; Vega and Kenyon-Mathieu, 2007). Another example is the traveling salesman problem, which admits multiple integer programming formulations (Orman and H. Williams, 2007; Öncan, Altınel, and Laporte, 2009). One can also formulate many problems using a graph-based representation (see Figure 4.1). Beyond learning combinatorial optimization solvers, other examples with multiple state-action representations include robotic applications with multiple sensing modalities such as third-person view demonstrations (Stadie, Abbeel, and Sutskever, 2017) and multilingual machine translation (Johnson et al., 2017).

In the context of policy learning, one natural question is how different state-action formulations impact learning and, more importantly, how learning can make use of multiple formulations. This is related to the co-training problem (Blum and Mitchell, 1998), where different feature representations of the same problem enable more effective learning than using only a single representation (Wan, 2009; Kumar and Daumé, 2011). While co-training has received much attention in classification tasks, little effort has been made to apply it to sequential decision making problems. One issue that arises in the sequential case is that some settings have completely separate state-action representations while others can share the action space.

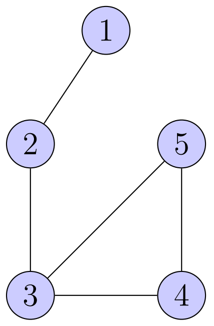
In this paper, we propose CoPiEr (co-training for policy learning), a meta-framework for policy co-training that can incorporate both reinforcement learning and imitation learning as subroutines. Our approach is based on a novel theoretical result that integrates and extends results from PAC analysis for co-training (Dasgupta, Littman, and McAllester, 2002) and general policy learning with demonstrations (Kang, Jie, and Feng, 2018). To the best of our knowledge, we are the first to formally extend the co-training framework to policy learning.

Our contributions can be summarized as:

- We present a formal theoretical framework for policy co-training. Our results include: 1) a general theoretical characterization of policy improvement, and 2) a specialized analysis in the shared-action setting to explicitly quantify the performance gap (i.e., regret) versus the optimal policy. These theoretical characterizations shed light on rigorous algorithm design for policy learning that can appropriately exploit multiple state-action representations.

- We present CoPiEr (co-training for policy learning), a meta-framework for policy co-training. We specialize CoPiEr in two ways: 1) a general mechanism for policies operating on different representations to provide demonstrations to each other, and 2) a more granular approach to sharing demonstrations in the shared-action setting.

- We empirically evaluate on problems in combinatorial optimization and discrete/continuous control. We validate our theoretical characterizations to identify when co-training can improve on single-view policy learning. We



$$
\begin{aligned}
&\max \; -\sum\nolimits_{i=1}^{5} x_i,\\
&\text{subject to:}\\
&x_1 + x_2 \geq 1,\\
&x_2 + x_3 \geq 1,\\
&x_3 + x_4 \geq 1,\\
&x_3 + x_5 \geq 1,\\
&x_4 + x_5 \geq 1,\\
&x_i \in \{0, 1\}, \forall i \in \{1, \cdots, 5\}
\end{aligned}
$$

Figure 4.1: Two ways to encode minimum vertex cover (MVC) problems. Left: policies learn to operate directly on the graph view to find the minimal cover set E. Khalil, Le Bodic, et al., 2016. Right: we express MVC as an integer linear program, then polices learn to traverse the resulting combinatorial search space, i.e., learn to branch-and-bound He, Daume III, and Eisner, 2014; J. Song, Lanka, Zhao, et al., 2018.

further showcase the practicality of our approach for the combinatorial optimization setting, by demonstrating superior performance compared to a wide range of strong learning-based benchmarks as well as commercial solvers such as Gurobi.

4.2 Background & Preliminaries

Markov Decision Process with Two State Representations. A Markov decision process (MDP) is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma, \mathcal{S}_T)$. Let $\mathcal{S}$ denote the state space, $\mathcal{A}$ the action space, $\mathcal{P}(s'|s,a)$ the (probabilistic) state dynamics, $r(s,a)$ the reward function, γ the discount factor and (optinal) $\mathcal{S}_T$ a set of terminal states where the decision process ends. We consider both stochastic and deterministic MDPs. An MDP with two views can be written as $\mathcal{M}^A = (\mathcal{S}^A, \mathcal{A}^A, \mathcal{P}^A, r^A, \gamma^A, \mathcal{S}_T^A)$ and $\mathcal{M}^B = (\mathcal{S}^B, \mathcal{A}^B, \mathcal{P}^B, r^B, \gamma^B, \mathcal{S}_T^B)$. To connect the two views, we make the following assumption about the ability to translate trajectories between the two views.

Assumption 1. *For a (complete) trajectory in $\mathcal{M}^A$, $\tau^A = (s_0^A, a_0^A, s_1^A, a_1^A, \cdots, s_n^A)$, there is a function $f_{A\to B}$ that maps τ^A to its corresponding (complete) trajectory τ^B in the other view $\mathcal{M}^B$: $f_{A\to B}(\tau^A) = \tau^B = (s_0^B, a_0^B, s_1^B, a_1^B, \cdots, s_m^B)$. The rewards for τ^A and τ^B are the same under their respective reward functions, i.e., $\sum_{i=0}^{n-1} r^A(s_i^A, a_i^A) = \sum_{j=0}^{m-1} r^B(s_j^B, a_j^B)$. Similarly, there is a function $f_{B\to A}$ that* *maps trajectories in $\mathcal{M}^B$ to $\mathcal{M}^A$ which preserves the total rewards.*

Note that in Assumption 1, the length of τ^A and τ^B can be different because of different state and action spaces.

Combinatorial Optimization Example. Minimum vertex cover (MVC) is a combinatorial optimization problem defined over a graph $G = (V, E)$. A cover set is a subset $U \subset V$ such that every edge $e \in E$ is incident to at least one $v \in U$. The objective is to find a U with the minimal cardinality. For the graph in Figure 4.1, a minimal cover set is $\{2, 3, 4\}$.

There are two natural ways to represent an MVC problem as an MDP. The first is graph-based (H. Dai, E. B. Khalil, et al., 2017) with the action space as V and the state space as sequences of vertices in V representing partial solutions. The deterministic transition function is the obvious choice of adding a vertex to the current partial solution. The rewards are -1 for each selected vertex. A terminal state is reached if the selected vertices form a cover.

The second way is to formulate an integer linear program (ILP) that encodes an MVC problem:

$$\begin{aligned}
&\max \; -\sum\nolimits_{v \in V} x_v, \\
&\text{subject to}: \\
&x_u + x_v \geq 1, \forall e = (u, v) \in E, \\
&x_v \in \{0, 1\}, \forall v \in V.
\end{aligned}$$

We can then use branch-and-bound (Land and Doig, 2010) to solve this ILP, which represents the optimization problem as a search tree, and explores different areas of a search tree through a sequence of branching operations. The MDP states then represent current search tree, and the actions correspond to which node to explore next. The deterministic transition function is the obvious choice of adding a new node into the search tree. The reward is 0 if an action does not lead to a feasible solution and is the objective value of the feasible solution minus the best incumbent objective if an action leads to a node with a better feasible solution. A terminal state is a search tree which contains an optimal solution or reaches a limit on the number of nodes to explore.

The relationship between solutions in the two formulations are clear. For a graph $G = (V, E)$, a feasible solution to the ILP corresponds to a vertex cover by selecting all the vertices $v \in V$ with $x_v = 1$ in the solution. This correspondence ensures the existence of mappings between two representations that satisfy Assumption 1.

Note that, despite the deterministic dynamics, solving MVC and other combinatorial optimization problems can be extremely challenging due to the very large state space. Indeed, policy learning for combinatorial optimization is a topic of active research E. Khalil, Le Bodic, et al., 2016; He, Daume III, and Eisner, 2014; J. Song, Lanka, Zhao, et al., 2018; Mirhoseini et al., 2017; Balunovic, Bielik, and Vechev, 2018.

Policy Learning. We consider policy learning over a distribution of MDPs. For instance, there can be a distribution of MVC problems. Formally, we have a distribution $\mathcal{D}$ of MDPs that we can sample from (i.e., $\mathcal{M} \sim \mathcal{D}$). For a policy π, we define the following terms:

$$\begin{aligned}
\eta(\pi, \mathcal{M}) &= \mathbb{E}_{\tau\sim\pi}[\sum_{i=0}^{n-1} \gamma^i r(s_i, a_i)], \\
J(\pi) &= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}}[\eta(\pi, \mathcal{M})], \\
Q_\pi(s, a) &= \mathbb{E}_{\tau\sim\pi}[\sum_{i=0}^{n-1} \gamma^i r(s_i, a_i) | s_0 = s, a_0 = a], \\
V_\pi(s) &= \mathbb{E}_{\tau\sim\pi}[\sum_{i=0}^{n-1} \gamma^i r(s_i, a_i) | s_0 = s], \\
A_\pi(s, a) &= Q_\pi(s, a) - V_\pi(s),
\end{aligned}$$

with η being the expected cumulative reward of an individual MDP $\mathcal{M}$, J the overall objective, Q the Q function, V the value function, and A the advantage function. The performance of two policies can be related via the advantage function (Schulman, Levine, et al., 2015; Kakade and Langford, 2002): $\eta(\pi', \mathcal{M}) = \eta(\pi, \mathcal{M}) + \mathbb{E}_{\tau\sim\pi'}[\sum_{i=0}^{n-1} \gamma^i A_\pi(s_i, a_i)]$. Based on Theorem 4 below, we can rewrite the final term with the occupancy measure, $\rho_\pi(s, a) = \mathbb{P}(\pi(s) = a) \sum_{i=0}^{\infty} \gamma^i \mathbb{P}(s_i = s|\pi)$.

Theorem 4. *(Theorem 2 of (U. Syed, Bowling, and Schapire, 2008)). For any policy π, it is the only policy that has its corresponding occupancy measure ρ_π, i.e., there is a one-to-one mapping between policies and occupancy measures. Specifically, $\mathbb{P}(\pi(s) = a) = \frac{\rho_\pi(s,a)}{\sum_{a'} \rho_\pi(s,a')}$.*

With slight notation abuse, define $\rho_\pi(s) = \sum_{i=0}^{\infty} \gamma^i \mathbb{P}(s_i = s|\pi)$ to be the state visitation distribution. In policy iteration, we aim to maximize:

$$\begin{aligned}
&\mathbb{E}_{\tau\sim\pi'}[\sum\nolimits_{i=0}^{n-1}\gamma^i A_\pi(s_i,a_i)],\\
&=\sum\nolimits_{i=0}^{n-1}\mathbb{E}_{s_i\sim\rho_{\pi'}(s)}[\mathbb{E}_{a_i\sim\pi'(s_i)}[\gamma^i A_\pi(s_i,a_i)]],\\
&\approx\sum\nolimits_{i=0}^{n-1}\mathbb{E}_{s_i\sim\rho_{\pi}(s)}[\mathbb{E}_{a_i\sim\pi'(s_i)}[\gamma^i A_\pi(s_i,a_i)]].
\end{aligned}$$

This is done instead of taking an expectation over $\rho_{\pi'}(s)$ which has a complicated dependency on a yet unknown policy π'. Policy gradient methods tend to use the approximation by using ρ_π which depends on the current policy. We define the approximate objective as:

$$\begin{aligned}
&\eta_\pi(\pi',\mathcal{M})\\
&=\eta(\pi,\mathcal{M})+\sum\nolimits_{i=0}^{n-1}\mathbb{E}_{s_i\sim\rho_{\pi}(s)}[\mathbb{E}_{a_i\sim\pi'(s_i)}[\gamma^i A_\pi(s_i,a_i)]],
\end{aligned}$$

and its associated expectation over $\mathcal{D}$ as $J_\pi(\pi')=\mathbb{E}_{\mathcal{M}\sim\mathcal{D}}[\eta_\pi(\pi',\mathcal{M})]$.

4.3 A Theory of Policy Co-training

In this section, we provide two theoretical characterizations of policy co-training. These characterizations highlight a trade-off in sharing information between different views, and motivates the design of our CoPiEr algorithm presented in Section 4.4.

We restrict our analysis to infinite horizon MDPs, and thus require a strict discount factor $\gamma < 1$. We show in our experiments that our CoPiEr algorithm performs well even in finite horizon MDPs with $\gamma = 1$. Due to space constraints, we defer all proofs to the appendix.

We present two theoretical analyses with different types of guarantees:

- Section 4.3 quantifies the **policy improvement** in terms of policy advantages and differences, and caters to policy gradient approaches.
- Section 4.3 quantifies the **performance gap** with respect to an optimal policy in terms of policy disagreements, which is a stronger guarantee than policy improvement. This analysis is restricted to the shared action space setting, and caters to learning reduction approaches.

General Case: Policy Improvement with Demonstrations

For an MDP $\mathcal{M}\sim\mathcal{D}$, consider the rewards of two policies with different views $\eta^A(\pi^A,\mathcal{M}^A)$ and $\eta^B(\pi^B,\mathcal{M}^B)$. If $\eta^A(\pi^A,\mathcal{M}^A) > \eta^B(\pi^B,\mathcal{M}^B)$, π^A performs better than π^B on this instance , and we could use the translated trajectory of π^A as a demonstration for π^B. Even when $J(\pi^A) > J(\pi^B)$, because J is computed in expectation over $\mathcal{D}$, π^B can still outperform π^A on some MDPs. Thus it is possible for the exchange of demonstrations to go in both directions.

Formally, we can partition the distribution $\mathcal{D}$ into two (unnormalized) parts $\mathcal{D}_1$ and $\mathcal{D}_2$ such that the support of $\mathcal{D}$, supp($\mathcal{D}$) $=$ supp($\mathcal{D}_1$) $\cup$ supp($\mathcal{D}_2$) and supp($\mathcal{D}_1$) $\cap$ supp($\mathcal{D}_2$) $= \emptyset$, where for an MDP $\mathcal{M} \in$ supp$(\mathcal{D}_1), \eta(\pi^A, \mathcal{M}^A) \geq \eta(\pi^B, \mathcal{M}^B)$ and for an MDP $\mathcal{M} \in$ supp$(\mathcal{D}_2), \eta(\pi^B, \mathcal{M}^B) > \eta(\pi^A, \mathcal{M}^A)$. By construction, we can quantify the performance gap as:

Definition 1.

$$\begin{aligned}\delta_1 &= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_1}[\eta(\pi^A, \mathcal{M}^A) - \eta(\pi^B, \mathcal{M}^B)] \geq 0,\\ \delta_2 &= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\eta(\pi^B, \mathcal{M}^B) - \eta(\pi^A, \mathcal{M}^A)] > 0.\end{aligned}$$

We can now state our first result on policy improvement.

Theorem 5. *(Extension of Theorem 1 in (Kang, Jie, and Feng, 2018)) Define:*

$$\begin{aligned}\alpha^A_{\mathcal{D}} &= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}}[\max_s D_{KL}(\pi^A(s)\|\pi'^A(s))],\\ \beta^B_{\mathcal{D}_2} &= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\max_s D_{JS}(\pi^B(s)\|\pi^A(s))],\\ \alpha^B_{\mathcal{D}} &= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}}[\max_s D_{KL}(\pi^B(s)\|\pi'^B(s))],\\ \beta^A_{\mathcal{D}_1} &= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_1}[\max_s D_{JS}(\pi^A(s)\|\pi^B(s))],\\ \epsilon^B_{\mathcal{D}_2} &= \max_{\mathcal{M}\in supp(\mathcal{D}_2)} \max_{s,a} |A_{\pi^B}(s,a)|,\\ \epsilon^A_{\mathcal{D}} &= \max_{\mathcal{M}\in supp(\mathcal{D})} \max_{s,a} |A_{\pi^A}(s,a)|,\\ \epsilon^A_{\mathcal{D}_1} &= \max_{\mathcal{M}\in supp(\mathcal{D}_1)} \max_{s,a} |A_{\pi^A}(s,a)|,\\ \epsilon^B_{\mathcal{D}} &= \max_{\mathcal{M}\in supp(\mathcal{D})} \max_{s,a} |A_{\pi^B}(s,a)|.\end{aligned}$$

Here D_{KL} & D_{JS} denote the Kullback-Leibler and Jensen-Shannon divergence respectively. Then we have:

$$\begin{aligned}J(\pi'^A) &\geq J_{\pi^A}(\pi'^A) - \frac{2\gamma^A(4\beta^B_{\mathcal{D}_2}\epsilon^B_{\mathcal{D}_2} + \alpha^A_{\mathcal{D}}\epsilon^A_{\mathcal{D}})}{(1-\gamma^A)^2} + \delta_2,\\ J(\pi'^B) &\geq J_{\pi^B}(\pi'^B) - \frac{2\gamma^B(4\beta^A_{\mathcal{D}_1}\epsilon^A_{\mathcal{D}_1} + \alpha^B_{\mathcal{D}}\epsilon^B_{\mathcal{D}})}{(1-\gamma^B)^2} + \delta_1.\end{aligned}$$

Compared to conventional analyses on policy improvement, the new key terms that determine how much the policy improves are the β's and δ's. The β's, which

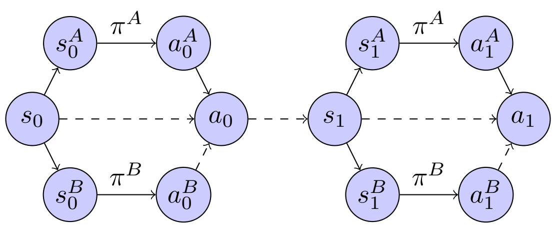


Figure 4.2: Co-training with shared action space.

quantify the maximal divergence between π^A and π^B, hinders improvement, while the δ's contribute positively. If the net contribution is positive, then the policy improvement bound is larger than that of conventional single view policy gradient. This insight motivates co-training algorithms that explicitly aim to minimize the β's.

One technicality is how to compute $D_{JS}(\pi^A(s)\|\pi^B(s))$ given that the state and action spaces for the two representations might be different. Proposition 6 ensures that we can measure the Jensen-Shannon divergence between two policies with different MDP representations.

Proposition 6. *For representations $\mathcal{M}^A$ and $\mathcal{M}^B$ of an MDP satisfying Assumption 1, the quantities* $\max_s D_{JS}(\pi^B(s)\|\pi^A(s))$ *and* $\max_s D_{JS}(\pi^A(s)\|\pi^B(s))$ *are well-defined.*

Minimizing $\beta^B_{\mathcal{D}_2}$ and $\beta^A_{\mathcal{D}_1}$ is not straightforward since the trajectory mappings between the views can be very complicated. We present practical algorithms in Section 4.4.

Special Case: Performance Gap from Optimal Policy in Shared Action Setting

We now analyze the special case where the action spaces of the two views are the same, i.e., $\mathcal{A}^A = \mathcal{A}^B$. Figure 4.2 depicts the learning interaction between π^A and π^B. For each state s, we can directly compare actions chosen by the two policies since the action space is the same. This insight leads to a stronger analysis result where we can bound the gap between a co-trained policy with an optimal policy. The approach we take resembles learning reduction analyses for interactive imitation learning.

For this analysis we focus on discrete action spaces with k actions, deterministic learned policies, and a deterministic optimal policy (which is guaranteed to exist (Puterman, 2014)). We reduce policy learning to classification: for a given state s,

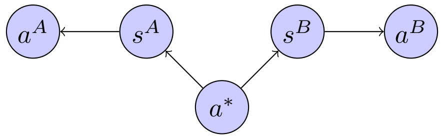


Figure 4.3: Graphical model encodes the conditional independence model.

the task of identifying the optimal action $\pi^*(s)$ is a classification problem. We build upon the PAC generalization bound results in (Dasgupta, Littman, and McAllester, 2002) and show that under Assumption 2, optimizing a measure of disagreements between the two policies leads to effective learning of π^*.

Assumption 2. *For a state s, its two representations s^A and s^B are conditionally independent given the optimal action $\pi^*(s)$.*

This assumption is common in analyses of co-training for classification (Blum and Mitchell, 1998; Dasgupta, Littman, and McAllester, 2002). Although this assumption is typically violated in practice (Nigam and Ghani, 2000), our empirical evaluation still demonstrates strong performance.

Assumption 2 corresponds to a graphical model describing the relationship between optimal actions and the state representations (Figure 4.3). The intuition is that, when we do not know $a^* = \pi^*(s)$, we should maximize the agreement between $a^A = \pi^A(s^A)$ and $a^B = \pi^B(s^B)$. By the data-processing inequality in information theory (Cover and Thomas, 2012), we know that $\mathbb{I}(a^A; a^*) \geq \mathbb{I}(a^A; a^B)$. In practice, this means that if a^A and a^B agree a lot, they must reveal substantial information about what a^* is. We formalize this intuition and obtain an upper bound on the classification error rate, which enables quantifying the performance gap. Notice that if we do not have any information from π^*, the best we can hope for is to learn a mapping that matches π^* up to some permutation of the action labels (Dasgupta, Littman, and McAllester, 2002). Thus we assume we have enough state-action pairs from π^* so that we can recover the permutation. In practice this is satisfied as we demonstrate in Section 4.5.

Formally, we connect the performance gap between a learned policy and an optimal policy with an empirical estimation on the disagreement in action choices among two co-trained policies. Let $\{\tau_i^A\}_{i=1}^m$ be sampled trajectories from π^A and $\{f_{A\to B}(\tau_i^A)\}_{i=1}^m$ be the mapped trajectories in $\mathcal{M}^B$. In $\{f_{A\to B}(\tau_i^A)\}_{i=1}^m$, let $N(a^A = i)$ be the number of times action i is chosen by π^A and $N = \sum_{i=1}^k N(a^A = i)$ be the total number of actions in one trajectory set. Let $N(a^B = i)$ be the number of times action i is chosen by π^B when going through the states in $\{f_{A\to B}(\tau_i^A)\}_{i=1}^m$ and $N(a^A = i, a^B = i)$ record when both actions agree on i.

We also require a measure of model complexity, as is common in PAC style analysis. We use $|\pi|$ to denote the number of bits needed to represent π. We can now state our second main result quantifying the performance gap with respect to an optimal policy:

Theorem 7. *If Assumption 2 holds for $\mathcal{M} \sim \mathcal{D}$ and a deterministic optimal policy π^*. Let π^A and π^B be two deterministic policies for the two representations. Define:*

$$\begin{aligned}
\hat{\mathbb{P}}(a^A = i \mid a^B = i) &= \frac{N(a^A = i, a^B = i)}{N(a^B = i)}, \\
\hat{\mathbb{P}}(a^A \neq i \mid a^B = i) &= \frac{N(a^A \neq i, a^B = i)}{N(a^B = i)}, \\
\epsilon_i(\pi^A, \pi^B, \sigma) &= \sqrt{\frac{\ln 2(|\pi^A| + |\pi^B|) + \ln{(2k/\sigma)}}{2N(a^B = i)}}, \\
\zeta_i(\pi^A, \pi^B, \sigma) &= \hat{\mathbb{P}}(a^A = i \mid a^B = i) \\
&\quad - \hat{\mathbb{P}}(a^A \neq i \mid a^B = i) - 2\epsilon_i(\pi^A, \pi^B, \sigma),
\end{aligned}$$

$$\begin{aligned}
b_i(\pi^A, \pi^B, \sigma) &= \frac{1}{\zeta_i(\pi^A, \pi^B, \delta)}(\hat{\mathbb{P}}(a^A \neq i \mid a^B = i) \\
&\quad + \epsilon_i(\pi^A, \pi^B, \sigma)), \\
\ell(s, \pi) &= \boldsymbol{1}(\pi(s) \neq \pi^*(s)), \\
\epsilon_A &= \mathbb{E}_{s\sim\rho_{\pi^A}}[\ell(s, \pi^A)].
\end{aligned}$$

Then with probability $1 - \sigma$:

$$\begin{aligned}
\epsilon_A &\leq \max_{j\in\{1,\cdots,k\}} b_j(\pi^A, \pi^B, \sigma), \\
\eta(\pi^A, \mathcal{M}^A) &\geq \eta(\pi^*, \mathcal{M}) - uT\epsilon_A,
\end{aligned}$$

where T is the time horizon and u is the largest one-step deviation loss compared with π^*.

To obtain a small performance gap compared to π^*, one must minimize ϵ_A, which measures the disagreement between π^A and π^*. However, we cannot directly estimate this quantity since we only have limited sample trajectories from π^*. Alternatively, we can minimize an upper bound, $\max_{j\in\{1,\cdots,k\}} b_j(\pi^A, \pi^B, \delta)$, which measures the maximum disagreement on actions between π^A and π^B and, importantly, can be estimated via samples. In Section 4.4, we design an algorithm that approximately minimizes this bound. The advantage of two views over a single view enables us to establish an upper bound on ϵ_A, which is otherwise unmeasurable.

4.4 The CoPiEr Algorithm

We now present practical algorithms motivated by the theoretical insights from Section 4.3. We start with a meta-algorithm named CoPiEr (Algorithm 5), whose important subroutines are EXCHANGE and UPDATE. We provide two concrete instantiations for the general case and the special case with a shared action space.

Algorithm 5 CoPiEr (Co-training for Policy Learning)

```
1: Input: A distribution $\mathcal{D}$ of MDPs, two policies $\pi^A, \pi^B$, mapping functions $f_{A\to B}, f_{B\to A}$
2: repeat
3:     Sample $\mathcal{M} \sim \mathcal{D}$, form $\mathcal{M}^A, \mathcal{M}^B$
4:     Run $\pi^A$ on $\mathcal{M}^A$ to generate trajectories $\{\tau_i^A\}_{i=1}^m$
5:     Run $\pi^B$ on $\mathcal{M}^B$ to generate trajectories $\{\tau_j^B\}_{j=1}^n$
6:     $\{\tau_i'^A\}, \{\tau_j'^B\} \leftarrow$ EXCHANGE$(\{\tau_i^A\}, \{\tau_j^B\})$
7:     $\pi^A \leftarrow$ UPDATE$(\pi^A, \{\tau_i^A\}, \{\tau_j'^A\})$
8:     $\pi^B \leftarrow$ UPDATE$(\pi^B, \{\tau_i^B\}, \{\tau_j'^B\})$
9: until Convergence
```

General Case

Algorithm 6 covers the general case for exchanging trajectories generated by the two policies. First we estimate the relative quality of the two policies from sampled trajectories (Lines 2-4 in Algorithm 6). Then we use the trajectories from the better policy as demonstrations for the worse policy on this MDP. This mirrors the theoretical insight presented in Section 4.3, where based on which sub-distribution an MDP is sampled from, the relative quality of the two policies is different.

For UPDATE, we can form a loss function that is derived from either imitation learning or reinforcement learning. Recall that we aim to optimize the β terms in Theorem 5, however it is infeasible to directly optimize them. So we consider a surrogate loss C (Line 2 of Algorithm 7) that measures the policy difference. In practice, we typically use behavior cloning loss as the surrogate.

Algorithm 6 EXCHANGE: General Case

```
1:  Input: Trajectories $\{\tau_i^A\}_{i=1}^m$ and $\{\tau_j^B\}_{j=1}^n$
2:  Compute estimate $\hat{\eta}(\pi^A, \mathcal{M}^A) = \frac{1}{m}\sum_{i=1}^m r(\tau_i^A)$
3:  Compute estimate $\hat{\eta}(\pi^B, \mathcal{M}^B) = \frac{1}{n}\sum_{j=1}^n r(\tau_j^B)$
4:  if $\hat{\eta}(\pi^A, \mathcal{M}^A) > \hat{\eta}(\pi^B, \mathcal{M}^B)$ then
5:      $\{\tau_i^{A\to B}\} \leftarrow \{f_{A\to B}(\tau_i^A)\}_{i=1}^m$
6:      $\{\tau_j^{B\to A}\} \leftarrow \emptyset$
7:  else
8:      $\{\tau_i^{A\to B}\} \leftarrow \emptyset$
9:      $\{\tau_j^{B\to A}\} \leftarrow \{f_{B\to A}(\tau_j^B)\}_{j=1}^n$
10: end if
11: return $\{\tau_i^{A\to B}\}, \{\tau_j^{B\to A}\}$
```

Algorithm 7 UPDATE

```
1:  Input: Current policy $\pi$, sampled trajectories from $\pi$, $\{\tau_i\}_{i=1}^m$ and demonstrations $\{\tau'_j\}_{j=1}^n$
2:  Form a loss function $\mathcal{L}(\pi) = \begin{cases} -\sum_{i=1}^m r(\tau_i) + \lambda C(\pi, \{\tau'_j\}_{j=1}^n), & \text{RL with IL loss} \\ \lambda C(\pi, \{\tau'_j\}_{j=1}^n), & \text{IL loss only} \end{cases}$
3:  Update $\pi \leftarrow \pi - \alpha \nabla \mathcal{L}(\pi)$
```

Algorithm 8 EXCHANGE: Special Case

```
1:  Input: Trajectories $\{\tau_i^A\}_{i=1}^m$ and $\{\tau_j^B\}_{j=1}^n$
2:  $D^{A\to B}$ = INTERACTIVE$(\{f_{B\to A}(\tau_j^B)\}_{j=1}^n, \pi^A)$
3:  $D^{B\to A}$ = INTERACTIVE$(\{f_{A\to B}(\tau_i^A)\}_{i=1}^m, \pi^B)$
4:  return $D^{A\to B}, D^{B\to A}$
```

Special Case: Shared Action Space

For the special case with a shared action space, we can collect more informative feedback beyond the trajectory level. Instead, we collect interactive state-level feedback, as is popular in imitation learning algorithms such as DAgger (Stéphane Ross, Gordon, and D. Bagnell, 2011) and related approaches W. Sun et al., 2017; Daumé III, Langford, and Marcu, 2009; Stéphane Ross and D. Bagnell, 2010; J. Song, Lanka, Zhao, et al., 2018; He, Daume III, and Eisner, 2014. Specifically, we can use Algorithms 8 & 9 to exchange actions in a state-coupled manner. This process is depicted in Figure 4.2, where π^A's visited states, s_0^A and s_1^A, are mapped to s_0^B and s_1^B, resulting in receiving π^B's actions, a_0^B and a_1^B, in the exchange.

Unlike the general case where information exchange is asymmetric, as Theorem 7 indicates, we aim to minimize policy disagreement. Both policies are simultaneously

Algorithm 9 INTERACTIVE

```
1: Input: Trajectories $\{\tau_i\}_{i=1}^m$, query policy $\pi$
2: $D = \emptyset$
3: for $i \leftarrow 1$ to $m$ do
4:     for each state $s \in \tau_i$ do
5:         $D \leftarrow D \cup \{(s, \pi(s))\}$
6:     end for
7: end for
8: return $D$
```

optimizing this objective, which requires both directions of information exchange (Lines 2-3 in Algorithm 8). The update step (Algorithm 7) is the same as the general case.

4.5 Experiments

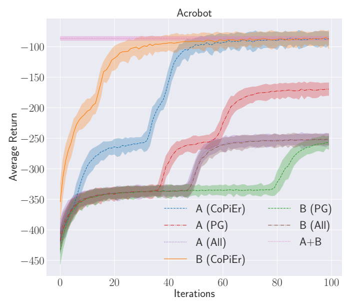
We now present empirical results on both the special and general cases of CoPiEr. We demonstrate the generality of our approach by applying three distinct combinations of policy co-training: reinforcement learning on both views (Section 4.5), reinforcement learning on one view and imitation learning on the other (Section 4.5), and imitation learning on both views (Section 4.5). Furthermore, our experiments on combinatorial optimization (Sections 4.5 & 4.5) demonstrate significant improvements over strong learning-based baselines as well as commercial solvers, and thus showcase the practicality of our approach. More details about the experiment setup can be found in the appendix.

Discrete & Continuous Control: Special Case with RL+RL

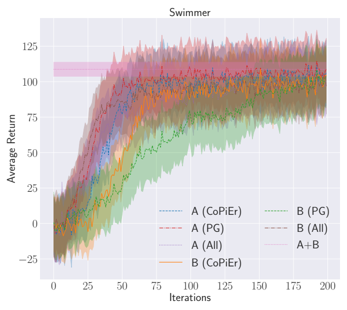
Setup. We conduct experiments on discrete and continuous control tasks with OpenAI Gym (Brockman et al., 2016) and Mujoco physical engine (Todorov, Erez, and Tassa, 2012). We use the garage repository (Duan et al., 2016) to run reinforcement learning for both views.

Two Views and Features. For each environment, states are represented by feature vectors, typically capturing location, velocity and acceleration. We create two views by removing different subsets of features from the complete feature set. Note that both views have the same underlying action space as the original MDP, so it is the special case covered in Section 4.4. We use interactive feedback for policy optimization.

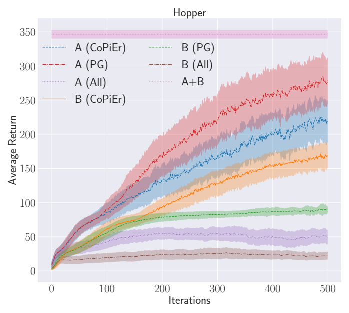
Policy Class. We use a feed-forward neural network with two hidden layers (64



(a) Acrobot Swing-up. A denotes removing the first coordinate in the state vector and B removing the second coordinate.



(b) Swimmer. A denotes removing all even index coordinates in the state vector and B removing all odd index ones.



(c) Hopper. A denotes removing all even index coordinates in the state vector and B removing all odd index ones.

Figure 4.4: Discrete & continuous control tasks. Experiment results are across 5 random seeded runs. Shaded area indicates ± 1 standard deviation.

& 32 units) and tanh activations as the policy class. For discrete actions, $\pi(s)$ outputs a soft-max distribution. For continuous actions, $\pi(s)$ outputs a (multivariate) Gaussian. For policy update, we use Policy Gradient (Sutton, McAllester, et al., 2000) with a linear baseline function (Greensmith, Bartlett, and Baxter, 2004) and define the loss function C in Algorithm 7 to be the KL-divergence between output action distributions.

Methods Compared. We compare with single view policy gradient, labelled as “A (PG)” and “B (PG)”, and with a policy trained on the union of the two views but test on two views separately, labelled as “A (All)” and “B (All)”. We also establish an upper bound on performance by training a model without view splitting (“A+B”). Each method uses the same total number of samples (i.e., CoPiEr uses half per view).

Results. Figure 4.4 shows the results. CoPiEr is able to converge to better or comparable solutions in almost all cases except for view A in Hopper. The poor performance in Hopper could be due to the disagreement between the two policies not shrinking enough to make Theorem 7 meaningful. As a comparison, at end of the training, the average KL-divergence for the two policies is about 2 for Hopper, compared with 0.23 for Swimmer and 0.008 for Acrobot. One possible cause for such large disagreement is that the two views have significance differences in difficulty for learning, which is the case for Hopper by noticing A (PG) and B (PG) have a difference in returns of about 190.

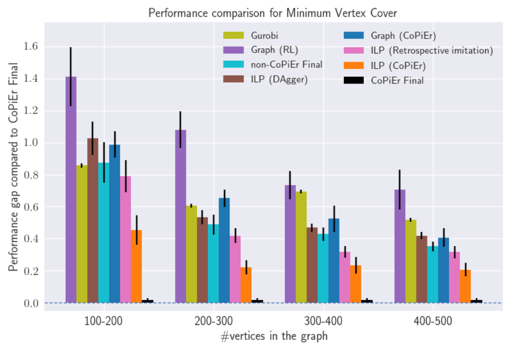


Figure 4.5: Comparison of CoPiEr with other learning-based baselines and a commercial solver, Gurobi. The y-axis measure relative gaps of various methods compared with CoPiEr Final. CoPiEr Final outperforms all the baselines. Notably, the gaps are significant because getting optimizing over large graphs is very challenging.

Minimum Vertex Cover: General Case with RL+IL

Setup. We now consider the challenging combinatorial optimization problem of minimum vertex cover (MVC). We use 150 randomly generated Erdős-Rényi (Erdős and Rényi, 1960) graph instances for each scale, with scales ranging {100-200, 200-300, 300-400, 400-500} vertices. For training, we use 75 instances, which we partition into 15 labeled and 60 unlabeled instances. We use the best solution found by Gurobi within 1 hour as the expert solution for the labeled set to bootstrap imitation learning. For each scale, we use 30 held-out graph instances for validation, and we report the performance on 45 test graph instances.

Views and Features. The two views are the graphs themselves and integer linear programs constructed from the graphs. For the graph view, we use DQN-based reinforcement learning (H. Dai, E. B. Khalil, et al., 2017) to learn a sequential vertex selection policy. We use `structure2vec` (H. Dai, B. Dai, and L. Song, 2016) to compute graph embeddings to use as state representations. For the ILP, we use imitation learning (He, Daume III, and Eisner, 2014) to learn node selection policy for branch-and-bound search. A node selection policy determines which node to explore next in the current branch-and-bound search tree. We use node-specific features (e.g., LP relaxation lower bound and objective value) and tree-specific features (e.g., integrality gap, and global lower and upper bounds) as our state representations. Vertex selection in graphs and node selection in branch-and-bound are different. So we use the general case algorithm in Section 4.4.

Policy Class. For the graph view, our policy class is similar to (H. Dai, E. B. Khalil, et al., 2017). In order to perform end-to-end learning of the parameters with labeled data exchanged between the two views, we use DQN (Mnih, Kavukcuoglu, Silver, Graves, et al., 2013) with supervised losses (Hester et al., 2018) to learn to imitate better demonstrations from the ILP view. For all our experiments, we determined the regularizer for the supervised losses and other parameters through cross-validation on the smallest scale (100-200 vertices). The graph view models are pre-trained with the labeled set using behavior cloning. We use the same number of training iterations for all the methods.

For the ILP view, our policy class consists of a node ranking model that prioritizes which node to visit next. We use RankNet (Burges, Renshaw, and Deeds, 1998) as the ranking model, instantiated using a 2-layer neural network with ReLU as activation functions. We implement our approach for the ILP view within the SCIP (Achterberg, 2009) integer programming framework.

Methods Compared. At test time, when a new graph is given, we run both policies and return the better solution. We term this practical version “CoPiEr Final" and measure other policies’ performance against it. We compare with single view learning baselines. For the graph view, we compare with RL-based policy learning over graphs (H. Dai, E. B. Khalil, et al., 2017), labelled as “Graph (RL)". And for the ILP view, we compare with imitation learning (He, Daume III, and Eisner, 2014) “ILP (DAgger)", retrospective imitation (J. Song, Lanka, Zhao, et al., 2018) “ILP (Retrospective Imitation)" and a commercial solver Gurobi (Gurobi Optimization, 2021). We combine “Graph (RL)" and “ILP (DAgger)" as non-CoPiEr (Final) by returning the better solution of the two. We also show the performance of the two policies in CoPiEr as standalone policies instead of combining them, labelled “Graph (CoPiEr)" and “ILP (CoPiEr)". ILP methods are limited by the same node budget in branch-and-bound trees.

Results. Figure 4.5 shows the results. We see that CoPiEr Final outperforms all baselines as well as Gurobi. Interestingly, it also performs much better than either standalone CoPiEr policies, which suggests that Graph (CoPiEr) is better for some instances while ILP (CoPiEr) is better on others. This finding validates combining the two views to maximize the benefits from both. For the exact numbers on the final performance, please refer to Appendix A.4.

Risk-aware Path Planning: General Case with IL+IL

Setup. We finally consider a practical application of risk-aware path planning (Ono and B. C. Williams, 2008). Given a start point, a goal point, a set of polygonal obstacles, and an upper bound of the probability of failure (risk bound), we must find a path, represented by a sequence of way points, that minimizes cost while limiting the probability of collision to within the risk bound. Details on the data generation can be found in the Appendix A.3. We report the performance evaluations on 50 test instances.

Views and Features. This problem can be formulated into a mixed integer linear program (MILP) as well as a quadratically constrained quadratic program (QCQP), both of which can be solved using branch-and-bound (Land and Doig, 2010; J. Linderoth, 2005). For each view, we learn a node selection policy for branch-and-bound via imitation learning. Feature representations are similar to ILP view in MVC experiment (Section 4.5). For the QCQP view, we use the state variables bounds along the trace for each node from the root in the branch and bound tree as an additional feature. Although the search framework is the same, because of the different nature of the optimization problem formulations, the state and action space are incompatible, and so we use the general case of CoPiEr. A pictorial representation of the two views is presented in Appendix A.2.

Policy Class. The policy class for both MILP and QCQP views is similar to that of ILP view in MVC (Section 4.5), and we learn node ranking models.

Methods Compared. Similar to MVC experiment, we compare other methods with "CoPiEr Final" which returns the better solution of the two. We use single view learning baselines, specifically those based on imitation learning (He, Daume III, and Eisner, 2014), "QCQP (DAgger)" and "MILP(DAgger)", and on retrospective imitation (J. Song, Lanka, Zhao, et al., 2018), "QCQP (Retrospective Imitation)" and "MILP (Retrospective Imitation)". Two versions of non-CoPiEr Final are presented, based on DAgger and Retrospective Imitation, respectively. Gurobi is also used to solve MILPs but it is not able to solve the QCQPs because they are non-convex.

Results. Figure 4.6 shows the results. Like in MVC, we again see that CoPiEr Final outperforms baselines as well as Gurobi. We also observe a similar benefit of aggregating both policies. The effectiveness of CoPiEr enables solving much larger problems than considered in previous work (J. Song, Lanka, Zhao, et al., 2018) (560 vs 1512 binary variables).

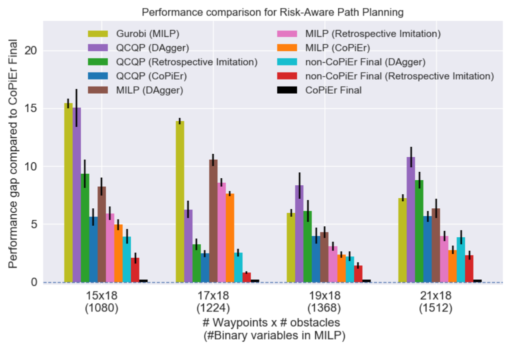


Figure 4.6: Comparison of CoPiEr with other learning-based baselines and a commercial solver, Gurobi. The y-axis measure relative gaps of various methods compared with CoPiEr Final. CoPiEr Final outperforms all the baselines. Notably, the scale of problems as measured by the number of integer variables far exceed previous state-of-the-art method (J. Song, Lanka, Zhao, et al., 2018).

Chapter 5

INCORPORATING EXISTING SOLVERS AS SUB-ROUTINES

Abstract In this chapter, we take a more holistic view of incorporating existing solvers. Compared with approaches presented in the previous two chapters, this view has the benefit of enabling learning-based methods to compete with state-of-the-art commercial solvers directly on wall-clock time.

We first present a learning approach building on the large neighborhood search (LNS) paradigm, which iteratively chooses a subset of variables to optimize while leaving the remainder fixed. The appeal of LNS is that it can easily use any existing solver as a subroutine, and thus can inherit the benefits of carefully engineered heuristic or complete approaches and their software implementations. We show that one can learn a good neighborhood selector using imitation and reinforcement learning techniques. Through extensive empirical validation in bounded-time optimization, we demonstrate that our LNS framework can significantly outperform compared to state-of-the-art commercial solvers such as Gurobi.

Then, we describe a machine learning approach to quickly solving Mixed Integer Programs (MIP) by learning to prioritize a set of decision variables, which we call pseudo-backdoors, for branching that results in faster solution times. Learning-based approaches have seen success in the area of solving combinatorial optimization problems by being able to flexibly leverage common structures in a given distribution of problems. Our approach takes inspiration from the concept of strong backdoors, which corresponds to a small set of variables such that only branching on these variables yields an optimal integral solution and a proof of optimality. Our notion of pseudo-backdoors corresponds to a small set of variables such that only branching on them leads to faster solve time (which can be solver dependent). A key advantage of pseudo-backdoors over strong backdoors is that they are much amenable to data-driven identification or prediction. Our proposed method learns to estimate the solver performance of a proposed pseudo-backdoor, using a labeled dataset collected on a set of training MIP instances. This model can then be used to identify high-quality pseudo-backdoors on new MIP instances from the same distribution. We evaluate our method on the generalized independent set problems and find that our approach can efficiently identify high-quality pseudo-backdoors. In addition, we compare our

learned approach against Gurobi, a state-of-the-art MIP solver, demonstrating that our method can be used to improve solver performance.

5.1 A General Large Neighborhood Search Framework for Solving Integer Linear Programs

The design of algorithms for solving hard combinatorial optimization problems remains a valuable and challenging task. Practically relevant problems are typically NP-complete or NP-hard. Examples include any kind of search problem through a combinatorial space, such as network designs (Du and Pardalos, 1998), mechanism design (De Vries and Vohra, 2003), planning (Ono and B. C. Williams, 2008), inference in graphical models (Wainwright, Jaakkola, and Willsky, 2005), program synthesis (Manna and Waldinger, 1971), verification (Bérard et al., 2013), and engineering design (Cui et al., 2006; Mirhoseini et al., 2017), amongst many others.

The widespread importance of solving hard combinatorial optimization problems has spurred intense research in designing approximation algorithms and heuristics for large problem classes, such as integer programming (Berthold, 2006; Fischetti and Lodi, 2010; Land and Doig, 2010) and satisfiability (L. Zhang and Malik, 2002; De Moura and Bjørner, 2008; Dilkina, Gomes, and Sabharwal, 2009). Historically, the design of such algorithms was done largely manually, requiring careful understandings of the underlying structure within specific classes of optimization problems. Such approaches are often unappealing due to the need to obtain substantial domain knowledge, and one often desires a more automated approach.

In recent years, there has been an increasing interest to automatically learn good (parameters of) algorithms for combinatorial problems from training data. The most popular paradigm, also referred to as "learning to search", aims to augment existing algorithmic templates by replacing hard-coded heuristic components with parameterized learnable versions. For example, this has been done in the context of greedy search for NP-hard graph problems (E. B. Khalil et al., 2017). However, greedy as well as most general purpose heuristic or local search algorithms are limited to combinatorial optimization problems, where the constraints are easy to satisfy, and hence are difficult to apply to domains with intricate side constraints. On the other hand, Integer Linear Programs (ILPs) are a widely applicable problem class that can encode a broad set of domains as well as large number and variety of constrains. Branch-and-bound, which is a complete search procedure, is the state-of-the-art approach to ILPs and has also been recently extensively researched through the lens of "learning to search" (He, Daume III, and Eisner, 2014; E. Khalil, Le Bodic, et al., 2016; E. B. Khalil et al., 2017; J. Song, Lanka, Zhao, et al., 2018; J. Song, Lanka, Yue, and Ono, 2020; Gasse et al., 2019). While this line of research has shown promise, it falls short of delivering practical impact, especially in improving wall-clock time. Practically improving the performance of branch-and-bound algorithms through learning is stymied by the need to either modify commercial solvers with limited access to optimized integration, or to modify open-source solvers such as SCIP (Achterberg, 2009), which is considerably slower than leading commercial solvers such as Gurobi and CPlex (usually by a factor of 10 or more) (Mittelmann, 2017; Optimization, 2019).

Motivated by the aforementioned drawbacks, we study how to design abstractions of large-scale combinatorial optimization problems that can leverage existing state-of-the-art solvers as a generic black-box subroutine. Our goal is to arrive at new approaches that can reliably outperform leading commercial solvers in wall-clock time, can be applicable to broad class of combinatorial optimization problems, and is amenable to data-driven design. We focus on solving integer linear programs (ILPs), which are a common way to represent many combinatorial optimization problems. We leverage the large neighborhood search (LNS) paradigm (Ahuja et al., 2002), an incomplete algorithm that iteratively chooses a subset of variables to optimize while leaving the remainder fixed. A major appeal of LNS is that it can easily use any existing solver as a subroutine, including ones that can handle general ILPs.

Our contributions can be summarized as:

- We propose a general LNS framework for solving large-scale ILPs. Our framework enables easy integration of existing solvers as subroutines, and does not depend on incorporating domain knowledge in order to achieve strong performance. In our experiments, we combine our framework with Gurobi, a leading commercial ILP solver.
- We show that, perhaps surprisingly, even using a *random* decision procedure within our LNS framework significantly outperforms Gurobi on many problem instances.
- We develop a learning-based approach that predicts a partitioning of the integer variables, which then serves as a learned decision procedure within our LNS framework. This procedure is effectively learning how to decompose

the original optimization problem into a series of sub-problems that can be solved much more efficiently using existing solvers.

- We perform an extensive empirical validation across several ILP benchmarks, and demonstrate superior wall-clock performance compared to Gurobi across all benchmarks. These results suggest that our LNS framework can effectively leverage leading state-of-the-art solvers to reliably achieve substantial speed-ups in wall-clock time.

5.2 Background on LNS

We now present our large neighborhood search (LNS) framework for solving integer linear programs (ILPs). LNS is a meta-approach that generalizes neighborhood search for optimization, and iteratively improves an existing solution by local search. As a concept, LNS has been studied for over two decades (Shaw, 1998; Ahuja et al., 2002; Pisinger and Ropke, 2010). However, previous work studied specialized settings with domain-specific decision procedures. For example, in Shaw, 1998, the definition of neighborhoods is highly specific to vehicle routing, and so the decision making of how to navigate the neighborhood is also domain-specific. We instead aim to develop a general framework that avoids requiring domain-specific structures, and whose decision procedures can be designed in a generic and automated way, e.g., via learning as described in Section 5.3. In particular, our approach can be viewed as a decomposition-based LNS framework that operates on generic ILP representations, as described in Section 5.2.

Background

Formally, let X be the set of all variables in an optimization problem and $\mathcal{S}$ be all possible value assignments of X. For a current solution $s \in \mathcal{S}$, a neighborhood function $N(s) \subset \mathcal{S}$ is a collection of candidate solutions to replace s, afterwards a solver subroutine is evoked to find the optimal solution within $N(s)$. Traditional neighborhood search approaches define $N(s)$ explicitly, e.g., the 2-opt operation in the traveling salesman problem (Dorigo, Birattari, and Stutzle, 2006). LNS defines $N(s)$ implicitly through a *destroy* and a *repair* method. A destroy method destructs part of the current solution while a repair method rebuilds the destroyed solution. The number of candidate repairments is potentially exponential in the size of the neighborhood, which explains the "large" in LNS.

In the context of solving ILPs, the LNS is also used as a local search heuristics for finding high quality incumbent solutions (Rothberg, 2007; Helber and Sahling, 2010; Hendel, 2018). The ways large neighborhoods are constructed are random (Rothberg, 2007), manually defined (Helber and Sahling, 2010) and via bandit algorithm selection from a pre-defined set (Hendel, 2018). Furthermore, these LNS approaches often require interface access to the underlying solver, which is often undesirable when designing frameworks that offer ease of deployment.

Recently, there has been some work on using learning within LNS (Hottung and Tierney, 2019; A. A. Syed et al., 2019). These approaches are designed for specific optimization problems, such as capacitated vehicle routing, and so are not directly comparable with our generic approach for solving ILPs. Furthermore, they often focus on learning the underlying solver (rather than rely on existing state-of-the-art solvers), which makes them unappealing from a deployment perspective.

Decomposition-based Large Neighborhood Search for Integer Programs

We now describe the details of our LNS framework. At a high level, our LNS framework operates on an ILP via defining decompositions of its integer variables into disjoint subsets. Afterwards, we can select a subset and use an existing solver to optimize the variables in that subset while holding all other variables fixed. The benefit of this framework is that it is completely generic to any ILP instantiation of any combinatorial optimization problem.

Without loss of generality, we consider the cost minimization objective. We first describe a version of LNS for integer programs based on decompositions of integer variables which is a modified version of the evolutionary approach proposed in Rothberg, 2007, outlined in Alg 10. For an integer program P with a set of integer variables X, we define a decomposition of the set X as a disjoint union $X_1 \cup X_2 \cup \cdots \cup X_k$. Assume we have an existing feasible solution S_X to P, we view each subset X_i of integer variables as a local neighborhood for search. We fix integers in $X \setminus X_i$ with their values in the current solution S_X and optimize for variable in X_i (referred as the `FIX_AND_OPTIMIZE` function in Line 3 of Alg 10). As the resulting optimization is a smaller ILP, we can use any off-the-shelf ILP solver to carry out the local search. In our experiments, we use Gurobi to optimize the sub-ILP. A new solution is obtained and we repeat the process with the remaining subsets.

Decomposition Decision Procedures. Notice that a different decomposition defines a different series of LNS problems and the effectiveness of our approach proceeds with a different decomposition for each iteration. The simplest implementation is to

Algorithm 10 `Decomposition-based LNS`

```
1: Input: an optimization problem $P$, an initial solutions $S_X$, a decomposition $X = X_1 \cup X_2 \cup \cdots \cup X_k$, a solver $F$
2: for $i = 1, \cdots, k$ do
3:     $S_X =$ FIX_AND_OPTIMIZE$(P, S_X, X_i, F)$
4: end for
5: return $S_X$
```

use a random decomposition approach, which we show empirically already delivers very strong performance. We can also consider learning-based approaches that learn a decomposition from training data, discussed further in Section 5.3.

5.3 Learning a Decomposition

In this study, we apply data-driven methods, such as imitation learning and reinforcement learning, to learn policies to generate decompositions for the LNS framework described in Section 5.2. We specialize a Markov decision process for our setting. For a combinatorial optimization problem instance P with a set of integer variables X, a state $s \in \mathcal{S}$ is a vector representing an assignment for variables in X, i.e., it is an incumbent solution. An action $a \in \mathcal{A}$ is a decomposition of X as described in Section 5.2. After running LNS through neighborhoods defined in a, we obtain a (new) solution s'. The reward $r(s, a) = J(s) - J(s')$ where $J(s)$ is the objective value of P when s is the solution. We restrict to finite-horizon task of length T so we set the discount factor γ to be 1.

Imitation Learning

In imitation learning, demonstrations (from an expert) serves as the learning signals. However, we do not have the access to an expert to generate good decompositions. Instead, we sample random decompositions and take the ones resulting in best objectives as demonstrations. This procedure is shown in Alg 11. The core of the algorithm is shown on Lines 7-12 where we repeatedly sample random decompositions and call the `Decomposition-based LNS` algorithm (Alg 10) to evaluate them. In the end, we record the decompositions with the best objective values (Lines 13-16).

Once we have generated a collection of good decompositions $\{D_i\}_{i=1}^{n}$, we apply two imitation learning algorithms. The first one is behavior cloning (Pomerleau, 1989). By turning each demonstration trajectory $D_i = (s_0, a_0, s_1, a_1, \cdots, s_{T-1}, a_{T-1})$ into a collection of state-action pairs $\{(s_0, a_0), (s_1, a_1), \cdots, (s_{T-1}, a_{T-1})\}$, we treat pol-

Algorithm 11 COLLECT_DEMOS

```
1: Input: a collection of optimization problems $\{P_i\}_{i=1}^n$ with initial solutions $\{S_i\}_{i=1}^n$, $T$
   the number of LNS iterations, $m$ the number of random decompositions to sample, $k$
   the number of subsets in a decompositon, $F$ a solver.
2: for $i = 1, \cdots, n$ do
3:     $best\_obj \leftarrow \infty$
4:     $best\_decomp \leftarrow None$
5:     for $j = 1, \cdots, m$ do
6:         $decomps \leftarrow []$
7:         for $t = 1, \cdots, T$ do
8:             $X \leftarrow$ RANDOM_DECOMPOSITION$(P_i, k)$
9:             $S_i \leftarrow$
10:            Decomposition-based LNS$(P_i, S_i, X, F)$
11:            $decomps.append(X)$
12:        end for
13:        if $J(S_i) < best\_obj$ then
14:            $best\_obj \leftarrow J(S_i)$
15:            $best\_decomp \leftarrow decomps$
16:        end if
17:    end for
18:    Record $best\_decomp$ for $P_i$
19: end for
20: return $best\_decompos$
```

Algorithm 12 Forward Training for LNS

```
1: Input: a collection of optimization problems $\{P_i\}_{i=1}^n$ with initial solutions $\{S_i\}_{i=1}^n$, $T$
   the time horizon, $m$ the number of random decompositions to sample, $k$ the number of
   subsets in a decompositon, $F$ a solver..
2: for $t = 1, \cdots, T$ do
3:     $\{D_i\}_{i=1}^n =$
4:     COLLECT_DEMOS$(\{P_i\}_{i=1}^n, \{S_i\}_{i=1}^n, 1, m, k, F)$
5:     $\pi_t =$ SUPERVISE_TRAIN$(\{D_i\}_{i=1}^n)$
6:     for $i = 1, \cdots, n$ do
7:         $X \leftarrow \pi_t(P_i, S_i)$
8:         $S_i \leftarrow$ Decomposition-based LNS$(P_i, S_i, X, F)$
9:     end for
10: end for
11: return $\pi_1, \pi_2, \cdots, \pi_T$
```

icy learning as a supervised learning problem. In our case, the action a is a decomposition which we represent as a vector. Each element of the vector indicates which subset X_i this particular variable belongs to. Thus, we reduce the learning problem to a supervised classification task.

Behavior cloning suffers from cascading errors (Stéphane Ross and D. Bagnell, 2010). We use the forward training algorithm (Stéphane Ross and D. Bagnell, 2010) to correct mistakes made at each step. We adapt the forward training algorithm for our use case and present it as Alg 12. The main difference with behavior cloning is the adaptive demonstration collection step on Line 4. In this case, we do not collect all demonstrations beforehand, instead, they are collected dependent on the predicted decompositions of previous policies.

Reinforcement Learning

For reinforcement learning, for simplicity, we choose to use REINFORCE (Sutton, McAllester, et al., 2000) which is a classical Monte-Carlo policy gradient method for optimizing policies. The goal is to find a policy π that maximizes $\eta(\pi) = \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)]$, the expected discounted accumulative reward. The policy π is normally parameterized with some θ. Policy gradient methods seek to optimize $\eta(\pi_\theta)$ by updating θ in the direction of: $\nabla_\theta \eta(\pi_\theta) = \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r(s_{t'}, a_{t'})]$. By sampling trajectories $(s_0, a_0, \cdots, s_{T-1}, a_{T-1}, s_T)$, one can estimate the gradient $\nabla_\theta \eta(\pi_\theta)$.

Featurization of an Optimization Problem

In this section, we describe the featurization of two classes of combinatorial optimization problems.

Combinatorial Optimization over Graphs. The first class of problems are defined explicitly over graphs as those considered in E. Khalil, H. Dai, et al., 2017. Examples include the minimum vertex cover, the maximum cut and the traveling salesman problems. The (weighted) adjacency matrix of the graph contains all the information to define the optimization problem so we use it as the feature input to a learning model.

General Integer Linear Programs. There are other classes of combinatorial optimization problems that do not originate from explicit graphs such as those in combinatorial auctions. Nevertheless, they can be modeled as integer linear programs. We construct the following incidence matrix A between the integer variables and the constraints. For each integer variable x_i and a constraint c_j, $A[i,j] = \text{coeff}(x_i, c_j)$ where $\text{coeff}(x_i, c_j)$ is the coefficient of the variable x_i in the constraint c_j if it appears in it and 0 otherwise.

Incorporating Current Solution. As outlined in Section 5.2, we seek to adaptively generate decompositions based on the current solution. Thus we need to include the solution in the featurization. Regardless of which featurization we use, the feature matrix has the same number of rows as the number of integer variables we consider, so we can simply append the variable value in the solution as an additional feature.

5.4 Emprical Validation for Learning-based LNS

We present experimental results on four diverse applications covering both combinatorial optimization over graphs and general ILPs. We discuss the design choices of crucial parameters in Section 5.4, and present the main results in Sections 5.4 & 5.4. Finally, we provide an empirical evaluation of a state-of-the-art learning to branch model with Gurobi in Section 5.4 to highlight the necessity of our proposed approach for practical impact.

Datasets & Setup

Datasets. We evaluate on 4 NP-hard benchmark problems expressed as ILPs. The first two, minimum vertex cover (MVC) and maximum cut (MAXCUT), are graph optimization problems. For each problem, we consider two random graph distributions, the Erdős-Rényi (ER) (Erdős and Rényi, 1960) and the Barabási-Albert (BA) (Albert and Barabási, 2002) random graph models. For MVC, we use graphs of size 1000. For MAXCUT, we use graphs of size 500. All the graphs are weighted and each vertex/edge weight is sampled uniformly from [0, 1] for MVC and MAXCUT, respectively. We also apply our method to combinatorial auctions (Leyton-Brown, Pearson, and Shoham, 2000) and risk-aware path planning (Ono and B. C. Williams, 2008), which are not based on graphs. We use the Combinatorial Auction Test Suite (CATS) (Leyton-Brown, Pearson, and Shoham, 2000) to generate auction instances from two distributions: regions and arbitrary. For each distribution, we consider two sizes: 2000 items with 4000 bids and 4000 items with 8000 bids. For the risk-aware path planning experiment, we use a custom generator to generate obstacle maps with 30 obstacles and 40 obstacles.

Learning a Decomposition. When learning the decomposition, we use 100 instances for training, 10 for validation and 50 for testing. When using reinforcement learning, we sample 5 trajectories for each problem to estimate the policy gradient. For imitation learning based algorithms, we sample 5 random decompositions and use the best one as demonstrations. All our experiment results are averaged over 5 random seeds.

Initialization. To run large neighborhood search, we require an initial feasible solution (typically quite far from optimal). For MVC, MAXCUT, and CATS, we initialize a feasible solution by including all vertices in the cover set, assigning all vertices in one set and accepting no bids, respectively. For risk-aware path planning, we initialize a feasible solution by running Gurobi for 3 seconds. This time is included when we compare wall-clock time with Gurobi.

Hyperparameter Configuration. We must set two parameters for our LNS approach. The first is k, the number of equally sized subsets to divide variables X into. The second is t, how long the solver runs on each sub-problem. A sub-ILP is still fairly large so solving it to optimality can take a long time, so we impose a time limit. We run a parameter sweep over the number of decompositions from 2 to 5 and time limit for sub-ILP from 1 second to 3 seconds. For each configuration of (k, t), the wall-clock time for one iteration of LNS will be different. For a fair comparison, we use the ratio Δ/T as the selection criterion for the optimal configuration, where Δ is the objective value improvement and T is the time spent. The configuration results are in Appendix B.1.

Benchmark Comparisons with Gurobi

We now present our main benchmark evaluations. We instantiate our framework in four ways:

- Random-LNS: using random decompositions
- BC-LNS: using a decomposition policy trained using behavior cloning
- FT-LNS: using a decomposition policy trained using forward training
- RL-LNS: using a decomposition policy trained using REINFORCE

We use Gurobi 9.0 as the underlying solver. For learned LNS methods, we sequentially generate 10 decompositions and apply LNS with these decompositions. We use the same time limit setting for running each sub-problem, as a result, the wall-clock among decomposition methods are very close.

We study the incomplete setting, where the goal is to find the best possible feasible solution within a bounded running time. When comparing using just Gurobi, we limit Gurobi's runtime to the longest runtime across all instances from our LNS methods. In other words, Gurobi's runtime is longer than all the decomposition based methods, which gives it more time to find the best solution possible.

	MVC BA 1000	MVC ER 1000	MAXCUT BA 500	MAXCUT ER 500
Gurobi	440.08 ± 1.26	482.15 ± 0.82	-3232.53 ± 16.61	-4918.07 ± 12.43
Random-LNS	433.59 ± 0.51	471.21 ± 0.36	$\mathbf{-3583.63 \pm 3.81}$	-5488.49 ± 6.60
BC-LNS	433.09 ± 0.53	$\mathbf{470.20 \pm 0.34}$	$\mathbf{-3584.90 \pm 4.02}$	$\mathbf{-5494.76 \pm 6.51}$
FT-LNS	$\mathbf{432.00 \pm 0.52}$	$\mathbf{470.04 \pm 0.37}$	$\mathbf{-3586.29 \pm 3.33}$	$\mathbf{-5496.29 \pm 6.69}$
RL-LNS	434.16 ± 0.38	471.52 ± 0.15	$\mathbf{-3584.70 \pm 1.49}$	-5481.57 ± 2.97

Table 5.1: Comparison of different LNS methods and Gurobi for MVC and MAX-CUT problems.

	CATS Regions 2000	CATS Regions 4000	CATS Arbitrary 2000	CATS Arbitrary 4000
Gurobi	-94559.9 ± 2640.2	-175772.9 ± 2247.89	-69644.8 ± 1796.9	-142168.1 ± 4610.0
Random-LNS	-99570.1 ± 790.5	-201541.7 ± 1131.1	-85276.6 ± 680.9	-170228.3 ± 1711.7
BC-LNS	$\mathbf{-101957.5 \pm 752.7}$	-207196.2 ± 1143.8	-86659.6 ± 720.2	$\mathbf{-172268.1 \pm 1594.8}$
FT-LNS	$\mathbf{-102247.9 \pm 709.0}$	$\mathbf{-208586.3 \pm 1211.7}$	$\mathbf{-87311.8 \pm 676.0}$	-169846.7 ± 5293.2

Table 5.2: Comparison of different LNS methods and Gurobi for CATS problems.

	30 Obstacles	40 Obstacles
Gurobi	0.7706 ± 0.23	0.7407 ± 0.13
Random-LNS	0.6487 ± 0.07	0.3680 ± 0.03
BC-LNS	$\mathbf{0.5876 \pm 0.07}$	$\mathbf{0.3502 \pm 0.07}$
FT-LNS	$\mathbf{0.5883 \pm 0.07}$	$\mathbf{0.3567 \pm 0.04}$

Table 5.3: Comparison of different LNS methods and Gurobi for risk-aware path planning problems.

Main Results. Tables 5.1, 5.2 and 5.3 show the main results. We make two observations:

- All LNS variants significantly outperform Gurobi (up to 50% improvement in objectives), given the same amount or less wall-clock time. Perhaps surprisingly, this phenomenon holds true even for Random-LNS.
- The imitation learning based variants, FT-LNS and BC-LNS, outperform Random-LNS and RL-LNS in most cases.

Overall, these results suggest that our LNS approach can reliably offer substantial improvements over state-of-the-art solvers such as Gurobi. These results also suggest that one can use learning to automatically design strong decomposition approaches, and we provide a preliminary qualitative study of what the policy has learned in Section B.2. It is possible that a more sophisticated RL method could further improve RL-LNS.

Per-Iteration Comparison. We use a total of 10 iterations of LNS, and it is natural to ask how the solution quality changes after each iteration. Figure 5.1 shows objective value progressions of variants of our LNS approach on three datasets. For

	MVC BA 1000	MVC ER 1000
Local-ratio	487.58 ± 1.16	498.20 ± 1.24
Best-LNS	$\mathbf{432.00 \pm 0.52}$	$\mathbf{470.04 \pm 0.37}$

	MAXCUT BA 500	MAXCUT ER 500
Greedy	-3504.79 ± 7.80	-5302.63 ± 17.59
Burer	$\mathbf{-3647.46 \pm 7.63}$	$\mathbf{-5568.18 \pm 13.47}$
De Sousa	-3216.86 ± 9.86	-4994.73 ± 10.60
Best-LNS	-3586.29 ± 3.33	-5496.29 ± 6.69

Table 5.4: (Left) Comparison between LNS with the local-ratio heuristic for MVC. (Right) Comparison between LNS with heuristics for MAXCUT.

	CATS Regions 2000	CATS Regions 4000	CATS Arbitrary 2000	CATS Arbitrary 4000
Greedy	-89281.4 ± 1296.4	-181003.5 ± 1627.5	-81588.7 ± 1657.6	-114015.9 ± 12313.8
LP Rounding	-87029.9 ± 876.66	-173004.1 ± 1688.9	-74545.1 ± 1365.5	-104223.1 ± 11124.5
Best-LNS	$\mathbf{-102247.9 \pm 709.0}$	$\mathbf{-208586.3 \pm 1211.7}$	$\mathbf{-87311.8 \pm 676.0}$	$\mathbf{-172268.1 \pm 1594.8}$

Table 5.5: Comparison between LNS with greedy and LP rounding heuristics for CATS.

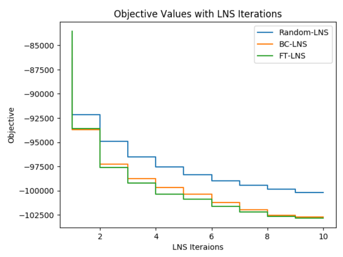
	Set Cover	Independence Set	Facility	CATS
GCNN (Gasse et al., 2019)	$1489.91 \pm 3.3\%$	$2024.37 \pm 30.6\%$	$563.36 \pm 10.7\%$	$114.16 \pm 10.3\%$
Gurobi	$\mathbf{669.64 \pm 0.55\%}$	$\mathbf{51.53 \pm 5.25\%}$	$\mathbf{39.87 \pm 3.91\%}$	$\mathbf{40.99 \pm 7.19\%}$

Table 5.6: Wall-clock comparison between learning to branch GCNN models (Gasse et al., 2019) and Gurobi.

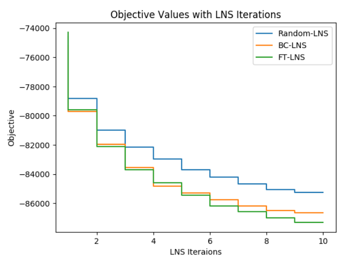
the two combinatorial auction datasets, BC-LNS and FT-LNS achieve substantial performance gains over Random-LNS after just 2 iterations of LNS, while it takes about 4 for the risk-aware path planning setting. These results show that learning a decomposition method for LNS can establish early advantages over using random decompositions.

Running Time Comparison. Our primary benchmark comparison limited all methods to roughly the same time limit. We now investigate how the objective values improve over time by performing 100 iterations of LNS. Figure 5.2 shows four representative instances. We see that FT-LNS achieves the best performance profile of solution quality vs. wall-clock.

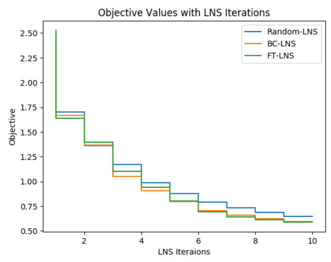
How Long Does Gurobi Need? Figure 5.2 also allows us to compare with the performance profile of Gurobi. In all cases, LNS methods find better objective values than Gurobi early on and maintain this advantage even as Gurobi spends significantly more time. Most notably, in Figure 5.2d, Gurobi was given 2 hours of wall-clock time, and failed to match the solution found by Random-LNS in just under 5 seconds (the time axis is in log scale).



(a) CATS with 2000 items and 4000 bids from regions distribution.

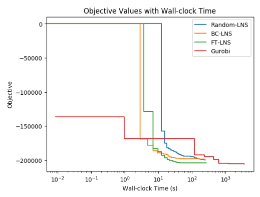


(b) CATS with 2000 items and 4000 bids from arbitrary distribution.

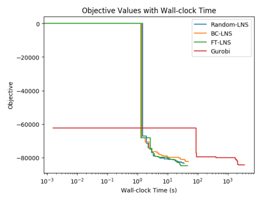


(c) Risk-aware path planning for 30 obstacles.

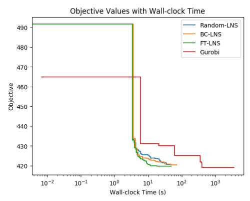
Figure 5.1: Improvements of objective values as more iterations of LNS are applied. In all three cases, imitation learning methods, BC-LNS and FT-LNS, outperform the Random-LNS.



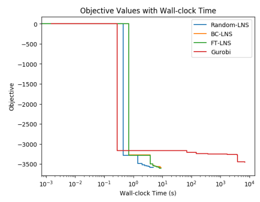
(a) CATS with 4000 items and 8000 bids from regions distribution.



(b) CATS with 2000 items and 4000 bids from arbitrary distribution.



(c) MVC over a Barabási-Albert random graph with 1000 vertices.



(d) MAXCUT over a Barabási-Albert random graph with 500 vertices.

Figure 5.2: We compare LNS methods on how the objective values improve as more wall-clock time is spent for some representative problem instances. We also include Gurobi in the comparison. All LNS methods find better solutions than Gurobi early on and it takes Gurobi between 2 to 10 times more time to match the solution quality. For MAXCUT (Fig 5.2d), after running for 2 hours, Gurobi is unable to match the quality of solution found by Random-LNS in 5 seconds.

Comparison with Domain-Specific Heuristics

We also compare with strong domain-specific heuristics for three classes of problems: MVC, MAXCUT, and CATS. We do not compare in the risk-aware path planning domain, as there are no readily available heuristics. Please refer to Appendix B.4 for descriptions on these heuristics.

Overall, we find that our LNS methods are competitive with specially designed heuristics, and can sometimes substantially outperform them. These results provide evidence that our LNS approach is a promising direction for the automated design of solvers that avoids the need to carefully integrate domain knowledge while achieving competitive or state-of-the-art performance.

	CATS Regions 2000
SCIP	-86578.38 ± 606.21
Random-LNS	-98944.90 ± 645.23
BC-LNS	-100513.84 ± 702.05
FT-LNS	-100913.77 ± 681.00

Table 5.7: LNS with SCIP as the ILP solver.

Table 5.4 (Left) summarizes results for MVC. The best LNS (FT-LNS) result outperforms by 11% on BA graphs and 6% on ER graphs. Table 5.4 (Right) shows results for MAXCUT. The heuristic in Burer et al.(Burer, Monteiro, and Y. Zhang, 2002) performs best for both random graph distributions, which shows that a specially designed heuristic can still outperform a general ILP solver. For CATS, our LNS approach outperforms both heuristics by up to 50% in objective values (Table 5.5).

Comparison with Learning to Branch Methods

Recently, there has been a surge of interest in applying machine learning methods within a branch-and-bound solver. Most prior work builds on the open-source solver SCIP (Achterberg, 2009), which is much slower than Gurobi and other commercial solvers (Mittelmann, 2017; Optimization, 2019). Thus, it is unclear how the demonstrated gains from these methods can translate to wall-clock time improvement. For instance, Table 5.6 shows a benchmark comparison between Gurobi and a state-of-the-art learning approach built on SCIP (Gasse et al., 2019) (reporting the 1-shifted geometric mean of wall-clock time on hard instances in their paper). These results highlight the large gap between Gurobi and a learning-augmented SCIP, which exposes the issue of designing learning approaches without considering how to integrate with existing state-of-the-art software systems (if the goal is to achieve good wall-clock time performance).

Use SCIP as the ILP Solver

Our LNS framework can incorporate any ILP solver to search for improvement over incumbent solutions. Our main experiments focused on Gurobi because it is a state-of-the-art ILP solver. Here, we also present results on using SCIP as the ILP solver. With the same setting as in Section 5.4 on the CATS Regions distribution with 2000 items and 4000 bids, the results are shown in Table 5.7. They are consistent with those when Gurobi is used as the ILP solver. Random-LNS significantly outperforms standard SCIP while learning-based methods (BC-LNS and FT-LNS) further improves upon Random-LNS.

5.5 Learning Pseudo-backdoors for Mixed Integer Programs

Mixed integer programs (MIPs) are widely used mathematical models for combinatorial optimization problems (Conforti, Cornuéjols, Zambelli, et al., 2014). An optimal solution to a MIP can be found by the branch-and-bound algorithm (Land and Doig, 2010), which systematically explores the solution space by building a search tree through branching on integer variables. It has been shown empirically that branching decisions have a significant impact on the solve time of MIPs (Achterberg and Wunderling, 2013). Recently, a series of papers have explored data-driven approaches to learning branching heuristics to improve MIP solve time (E. Khalil, Le Bodic, et al., 2016; Gasse et al., 2019). As a branch-and-bound algorithm builds the search tree sequentially, the total solve time of a MIP is proportional to the size of the final search tree. As a result, one can reduce solve time by producing a more compact search tree. The concept of backdoors is one structural property related to the search tree size of combinatorial problems. Backdoors are first introduced in (R. Williams, Gomes, and Selman, 2003) in the context of Boolean satisfiability (SAT) problems. Weak backdoors are defined to be a small subset of variables that satisfy the following property: there exists an assignment to this subset of variables such that the remaining SAT formula can be solved in polynomial time. In a "strong backdoor" any setting of the backdoor variables leads to a poly-time solvable subproblem. Given a backdoor, one can speed up SAT solving by restricting standard backtrack search only on variables in the backdoor. Later, backdoors are generalized to combinatorial optimization and MIPs (Dilkina, Gomes, Malitsky, et al., 2009), where a strong MIP backdoor is a subset of integer variables such that only branching on them yields an optimal integral solution and a certificate of optimality. Similarly to SAT, solve time speedup in MIPs has been observed by prioritizing "backdoors" in branching (Fischetti and Monaci, 2011). Technically, the prioritized variables do not satisfy the definition of backdoors given in (Dilkina, Gomes, Malitsky, et al., 2009), and to make this distinction clear we will refer to a subset of integer variables as a *pseudo-backdoor* if prioritizing branching on them leads to faster solve time compared with the default setting for a MIP solver.

In this extended abstract, we introduce a data-driven approach to predicting pseudo-backdoors for distributions of MIPs by learning a ranking model to identify which subset of variables is most likely to be a pseudo-backdoor and a classification model to decide whether to use a candidate pseudo-backdoor or just use the default solver. If the classifier decides to use the pseudo-backdoor, we prioritize a MIP solver to branch on variables in the pseudo-backdoor above other integral decision variables.

We represent MIPs as bipartite graphs with different node-features for variables and constraints as in (Gasse et al., 2019), and use graph attention networks (Veličković et al., 2018) with pooling to learn both models. We conduct empirical evaluations on the generalized independence set problem (GISP) (Colombi, Mansini, and M. Savelsbergh, 2017) and show that our models achieve overall improvements over Gurobi with its default setting.

5.6 Problem Statement for Learning Pseudo-backdoors

Our goal in finding pseudo-backdoors is to quickly solve MIPs. In a MIP we are asked to find real-valued settings for n decision variables $x \in \mathbb{R}^n$, which maximize a linear objective function $c^T x$, subject to m linear constraints $Ax \leq b$, and with a subset $\mathcal{I} \subseteq [n]$ of the decision variables required to be integral $x_i \in \mathbb{Z}\,\forall i \in \mathcal{I}$. Overall the problem can be written as

$$\begin{array}{ll} \max_x & c^T x \\ \text{subject to} & Ax \leq b \\ & x_i \in \mathbb{Z} \;\; \forall i \in \mathcal{I}. \end{array}$$

Given a MIP problem, specified by $P = (c, A, b, \mathcal{I})$, our goal is to find a pseudo-backdoor subset $\mathcal{B} \subseteq \mathcal{I}$, of the integral decision variables such that prioritizing branching on these decision variables yields fast solve times. We consider a distributional setting of MIP solving where we are given a training distribution of instances and want to train a pseudo-backdoor selection model that performs well on unseen instances from the same distribution.

5.7 Learning Pseudo-Backdoors

On a high level, our approach utilizes two learned models: a scoring model that scores subsets of integer variables according to how their likelihood of being pseudo-backdoors and a classifier that decides whether to use a predicted subset in the actual MIP solving. The intuition of including the second model is that some MIPs do not admit a small pseudo-backdoor in practice. For them, it is better to run a solver in its default setting.

Figure 5.3 illustrates our method. At test time, given a new MIP instance, we randomly sample subsets of integer variables according to their LP fractionality as in (Dilkina, Gomes, Malitsky, et al., 2009), scoring the sampled subsets, and taking the subset with the highest score as the predicted pseudo-backdoor. Then we use the classifier to decide whether to use the predicted pseudo-backdoor in a MIP solver

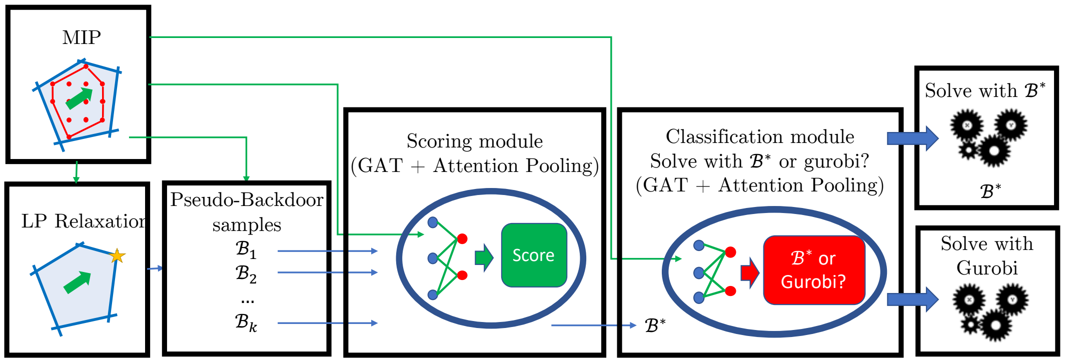

MIP
LP Relaxation
Pseudo-Backdoor samples
$\mathcal{B}_1$
$\mathcal{B}_2$
...
$\mathcal{B}_k$
Scoring module (GAT + Attention Pooling)
Score
$\mathcal{B}^*$
Classification module Solve with $\mathcal{B}^*$ or gurobi? (GAT + Attention Pooling)
$\mathcal{B}^*$ or Gurobi?
Solve with $\mathcal{B}^*$
$\mathcal{B}^*$
Solve with Gurobi

Figure 5.3: The pseudo-backdoor deployment pipeline visualizes the different components used for solving a single MIP instance with the two learned models, the scoring module $S(P, \mathcal{B}; \theta_S)$ and the classification module $C(P, \mathcal{B}; \theta_C)$. First k pseudo-backdoor sets of decision variables $\mathcal{B}_1, \ldots, \mathcal{B}_k$ are sampled according to the decision variables' LP fractionality. These candidate pseudo-backdoor sets are ranked according to the scoring module $S(P, \mathcal{B}; \theta_S)$ to predict the best pseudo-backdoor $\mathcal{B}^*$. The classification module then determines whether to run the solver using $\mathcal{B}^*$ or not based on the predicted pseudo-backdoor success $C(P, \mathcal{B}^*; \theta_C)$.

would result in faster solve time than the default setting. If the answer is positive, we assign higher branching priorities to those integer variables than the rest; otherwise, we run the solver with its default setting.

Concretely, the score model $S(P, \mathcal{B}; \theta_S)$ is parametrized by neural network parameters θ_S which takes as input the MIP specification P, and a candidate subset $\mathcal{B}$, then predicts a score that characterizes if $\mathcal{B}$ is a good pseudo-backdoor. The classifier $C(P, \mathcal{B}; \theta_C)$ is parametrized by neural network parameters θ_C which takes as input the MIP specification P, and a candidate subset $\mathcal{B}$, then predicts whether the prioritizing $\mathcal{B}$ in branching would produce a smaller runtime compared to running the solver.

Learning the Score Model

We train the score model S by learning to rank subsets of integer variables based on their quality as pseudo-backdoors. For a MIP P and two subsets of integer variables $\mathcal{B}_1, \mathcal{B}_2$ of P, we compute score estimates $s_1 = S(P, \mathcal{B}_1; \theta), s_2 = S(P, \mathcal{B}; \theta)$. Additionally, we compute a ranking label y which is -1 if $\mathcal{B}_1$ leads to a smaller runtime, and 1 otherwise. We then compute the marginal ranking loss (Tsochantaridis et al., 2005) as $\text{loss}(s_1, s_2, y) = \max(0, -y(s_1 - s_2) + m)$ for a given margin value m. The ranking loss allows the model to focus on distinguishing between relative performance rather than accurately modeling the absolute performance.

We want to ensure that S yields predictions that are invariant to changes that shouldn't modify the solver behavior such as permutations of variable labels. As a result, we consider a bipartite graph representation of a MIP as in (Gasse et al., 2019). This representation has two sets of nodes, one for variables and the other for constraints. There is one variable node for each decision variable and one constraint node for each constraint. Each variable node contains information such as the variable's objective coefficient, and root LP status. Each constraint node contains information such as the right hand side constant b_j, root LP dual variables, and sense ($\leq, \geq$ or $=$). We use the same set of features as in (Gasse et al., 2019). To form the bipartite graph, we add an edge between a variable i and a constraint j if variable i appears in constraint j, i.e., $A_{ij} \neq 0$ in the constraint matrix. The coefficient A_{ij} is encoded as an edge attribute. To represent a candidate pseudo-backdoor set $\mathcal{B}$ and retain the permutation invariance afforded by the graph representation, we consider a binary encoding of $\mathcal{B}$ by including an additional feature for each decision variable node which takes a value of 1 if the variable is in $\mathcal{B}$ and 0 otherwise. Encoding the input $(P, \mathcal{B})$ as a graph now allows us to leverage state-of-the-art techniques in making predictions on graphs which are amenable to variable input graph sizes and exhibit permutation invariance.

We leverage the Graph Attention Network (Veličković et al., 2018) where the nodes of the input graph are embedded by aggregating messages at a given node from its neighbors. Once we have several iterations of message passing along the edges of the bipartite graph, we aggregate all the node embeddings x_i using global attention pooling (Li et al., 2016) to yield a single feature vector representing the whole graph. Performing these steps of obtaining node embeddings followed by aggregation across the entire MIP enables us to use the same network architecture for MIPs of various sizes. Now that we have a fixed-length representation of the MIP, we feed it into a feedforward neural network to produce a scalar output as the score.

Learning the Classifier Model

For a given MIP instance, it may be difficult to sample valid pseudo-backdoors. As a result, we learn a subsequent classifier to determine whether to use the candidate subset or simply use a MIP solver in its default setting. The classifier has the same architecture as the scoring model, taking as input the bipartite graph representation of the MIP P and a candidate subset $\mathcal{B}$, and outputting a scalar value. However, in this module instead of ranking we perform binary classification. Thus, the last layer score is fed through a sigmoid activation function to get an output in the range of

$[0, 1]$ and the final binary output is obtained at a threshold of 0.5.

To generate training data, for every MIP instance in the training distribution, we compute the solve time for a solver with its default setting and for using the candidate pseudo-backdoor suggested by the previous score model. We then label the MIP instance and pseudo-backdoor according to whether the pseudo-backdoor results in a faster solve time. Finally, we compute a loss for this classification module as a binary cross-entropy loss between the model outputs and the labels.

5.8 Experiment Results for Learning Pseudo-backdoors

Many real-world setting require solving a homogeneous family of problems, where instances share similar structures, differing slightly in the problem size or numerical coefficients. We evaluate the two components of our proposed method on problem instances drawn from three hardness settings of the Generalized Independent Set Problem (GISP) (Hochbaum and Pathria, 1997).

We run data collection and MIP solving on a cluster of five identical 32-core machines with Intel 2.1 GHz processors and 264 GB of memory. We use the Python API of Gurobi 9.1 (Gurobi Optimization, 2021) to perform MIP data collection. For model evaluation, we integrate our two models with Gurobi by setting the branching priority of integer variables in a pseudo-backdoor to 1. To enable parallelization, we run Gurobi in the single-threaded mode without other modifications. The deep learning models are trained on a machine with four GeForce GTX 1080 Ti using Pytorch (Paszke et al., 2019).

Generalized Independent Set Problem

The Generalized Independent Set Problem (GISP) is a graph optimization problem initially proposed for forestry management (Hochbaum and Pathria, 1997). The input consists of a graph $G(V, E)$, a subset of removable edges $E' \subseteq E$, revenues for each vertex upon selection and costs for each removable edge upon deletion. The problem asks to select a subset of the vertices and remove a subset of removable edges that maximize the net profit of total vertex revenues minus total edge costs subject to the constraint that the selected vertices must be independent, i.e., pairs of selected vertices should not share an edge, in the induced subgraph after E' are removed from G.

We evaluate performance on randomly-generated GISP instances with varying parameters to yield different hardness settings. To generate a single instance we randomly generate an Erdős-Rényi graph (Erdős and Rényi, 1960) with n nodes and edge probability p. Edges are randomly added into removable edges E' with probability α. Finally, edges have fixed costs of c, and nodes have fixed rewards of r. We vary the problem hardness by varying the graph node count between 125 (easy), 150 (medium), and 175 (hard) nodes. We fix edge probabilities p to 0.3, removable edge probability α to 0.25, edge cost c to 1, and node reward r to 100.

For each hardness setting, we generate 300 random instances in total: 100 instances for training and validating the score model, 100 for training and validating the classifier model, and the final 100 for testing the two models. We denote the scoring dataset as $\mathcal{D}_s$, the classification dataset as $\mathcal{D}_c$, and the test dataset as $\mathcal{D}_t$.

Data Generation

To train our score and classifier models, we randomly sample subsets of integer variables for each training instance proportional to their LP-based fractionality as in (Dilkina, Gomes, Malitsky, et al., 2009). For a MIP instance, the method first computes the solution to its linear programming relaxation, and then randomly samples subsets of the integral decision variables to be included in the candidate pseudo-backdoor based on the fractionality, or how far away the value of a decision variable is to being integral, with less-integral decision variables having higher weight for entering into the backdoor. In (Dilkina, Gomes, Malitsky, et al., 2009), they found that many instances were solved with a relatively small backdoor size. So we randomly sample subsets of size $p \cdot |\mathcal{I}|$ for $p = 0.01$, i.e., we expect a pseudo-backdoor to contain 1% of the integer variables. We then collect runtime statistics for all the sampled subsets by running the MIP solver with higher branching priority on decision variables in the subsets.

For each MIP in our distribution, we collect performance metrics on 50 random subsets sampled according to LP fractionality (Dilkina, Gomes, Malitsky, et al., 2009). In total this yields a dataset of 15,000 pseudo-backdoors on which we will train, validate, and test our models on.

Model Training

For a given hardness setting, we train the score model on the 100 instances in $\mathcal{D}_s$. For each instance, we use the learning to rank formulation to train a score model that selects the best subset (with the fastest runtime) among the 50 random samples. Given the trained score model, we run it to score the 50 random subsets for each problem instance in $\mathcal{D}_c$ to produce a predicted pseudo-backdoor for each instance.

Then we compare the runtime using the predicted pseudo-backdoor with that by Gurobi to generate labels for training the classifier model. Finally, we evaluate the pipeline on the test set $\mathcal{D}_t$, following the pipeline illustrated in Figure 5.3 by identifying the best pseudo-backdoor with the score model, and then determining whether to use the selected pseudo-backdoor or standard Gurobi with the classifier model.

Main Results

We present testing results on the 100 MIPs from $\mathcal{D}_t$ for three hardness settings in Table 5.8. To understand the impact of using the two components of the model, we evaluate the standalone score model (scorer) in addition to the full pipeline (scorer + cls) with both the score model and the classifier model. We evaluate both absolute model performance in terms of runtime in seconds, and runtime win / tie / loss over Gurobi.

As we can see in Table 5.8, the score model alone performs well on many instances. It outperforms Gurobi on the easy and hard distributions in terms of having faster average runtimes and faster runtimes at different percentiles. The scoring module has 6% faster runtimes on both easy and hard instances; however, it has 57% slower runtimes on the medium instances. Furthermore, we can see from the win / loss rate against Gurobi that the score model alone is able to outperform Gurobi on a large number of the MIP instances, winning on 61 instances for easy, and 47 instances for hard. Indeed, the score model is able to solve 41 instances faster than gurobi on the medium difficulty instances, demonstrating that while it has poorer performance on average, it has potential for yielding fast solve times on many instances. Additionally, we can see that the score model alone has much higher variability than Gurobi on the medium instances. This undesired property further motivates the inclusion of the classifier model to improve the overall performance.

The scorer + cls model outperforms Gurobi across all three different MIP distributions in terms of the time distribution for MIP solving. On average, it performs well across distributions of instances, having the lowest solve times on average and at different quantiles. The score + cls pipeline outperforms gurobi by $8\%, 2\%, and 8\%$ on average for easy, medium, and hard instances respectively. In terms of win / tie / loss, we can see that this is partially due to reduced losses against Gurobi compared with only using the score model while retaining a large proportion of its wins. Furthermore, we can see that the variability on the medium difficulty instances is

Table 5.8: Runtime comparison in seconds of standard gurobi (grb), the score model (scorer), and the score model with subsequent classification (scorer+cls) across 3 hardness settings of gisp. In addition to mean and standard deviation of the runtimes, we report the 25th, 50th, and 75th percentiles of the runtimes across the MIP instances to provide further information about model performance at different points of the distribution. Finally, we report win / tie / loss metrics across the 100 test instances for the given models against Gurobi. Note that the scorer + classification module ties with Gurobi when it predicts to use Gurobi rather than the suggested psuedo-backdoor.

dataset	solver	mean	stdev	25 pct	median	75 pct	win / tie / loss vs grb
gisp easy	grb	108	**34**	79	109	128	0 / 100 / 0
gisp easy	scorer	101	41	71	95	126	61 / 0 / 39
gisp easy	scorer + cls	**99**	40	**69**	**93**	**119**	51 / 35 / 14
gisp medium	grb	611	**182**	488	580	681	0 / 100 / 0
gisp medium	scorer	960	755	515	649	915	41 / 0 / 59
gisp medium	scorer + cls	**601**	247	**481**	**568**	**663**	24 / 70 / 6
gisp hard	grb	2533	939	1840	2521	2976	0 / 100 / 0
gisp hard	scorer	2373	**855**	1721	2262	2926	47 / 0 / 53
gisp hard	scorer + cls	**2326**	**855**	**1654**	**2215**	**2866**	47 / 27 / 26

greatly reduced to be comparatively more on par with Gurobi than only the score model as well.

Chapter 6

LEARNING SURROGATES FOR OPTIMIZATION

Abstract Optimization algorithms employ different heuristic functions to guide the search. Computation of these heuristic functions can be expensive. Given historical data of the heuristic values, we can fit a machine learning model to estimate them from features of optimization problems. In this way, we reduce the computation time to simple feedforward computations if neural networks are used. In this chapter, we present two projects that utilize this idea.

In the first project (Section 6.1), we learn to estimate a safety cost called Approximate Clearance Evaluation (ACE) (Otsu et al., 2020) which is used in the Enhanced Navigation (ENav) (Toupet et al., 2020) library to plan paths for the Perseverance Rover in the Mars 2020 mission (Williford et al., 2018). We show that a deep convolutional neural network can predict the ACE costs with high accuracy and it enables us to plan paths faster without compromising on safety.

In the second project (Section 6.4), we further equip the surrogate with additional regularization properties, e.g., submodularity. This is motivated by cases where we know structures about the heuristic functions. In particular, we focus on a class of combinatorial problems that can be solved via submodular maximization (either directly on the objective function or via submodular surrogates). We introduce a data-driven optimization framework based on the *submodular-norm* loss, a novel loss function that encourages the resulting objective to exhibit *diminishing returns*. Our framework outputs a surrogate objective that is efficient to train, approximately submodular, and can be made permutation-invariant. The latter two properties allow us to prove strong approximation guarantees for the learned greedy heuristic. Furthermore, our model is easily integrated with modern deep imitation learning pipelines for sequential prediction tasks. We demonstrate the performance of our algorithm on a data-driven protein engineering task.

6.1 Learning Safety Surrogate for the Perseverance Rover

The Mars 2020 mission Williford et al., 2018 and its Perseverance Rover will use the Enhanced Navigation (ENav) library Toupet et al., 2020 to plan paths on the Martian surface. ENav takes as input stereo imagery, maintains a 2.5D heightmap describing the terrain, and chooses the best maneuver to safely move the rover toward the global goal. ENav uses the Approximate Clearance Evaluation (ACE) algorithm Otsu et al., 2020 to evaluate a sorted list of paths for safe traversal. Running the ACE algorithm on dozens of rover poses along hundreds of candidate rover paths represents a significant computational burden, especially if the list is sorted poorly and many paths fail the ACE check, which is more likely in complex and challenging terrain.

In this project, we train a machine learning (ML) classifier to infer ACE values to more effectively sort the rover paths before the ACE evaluation step. By incorporating the ML classifier into the ranking process, but still checking paths for safety with ACE, we show how ML can be integrated into ENav without sacrificing safety requirements. We present our results for various experiments and describe how each heuristic affected the performance of ENav in Monte Carlo simulations across multiple terrains (Section 6.3). We show that integrating the heuristics improved path efficiency, greatly reduced ACE evaluations, computation time, and the likelihood of "overthinking" each planning cycle, and maintained or improved success rates compared to the baseline performance.

6.2 Method for Learning Safety Surrogate

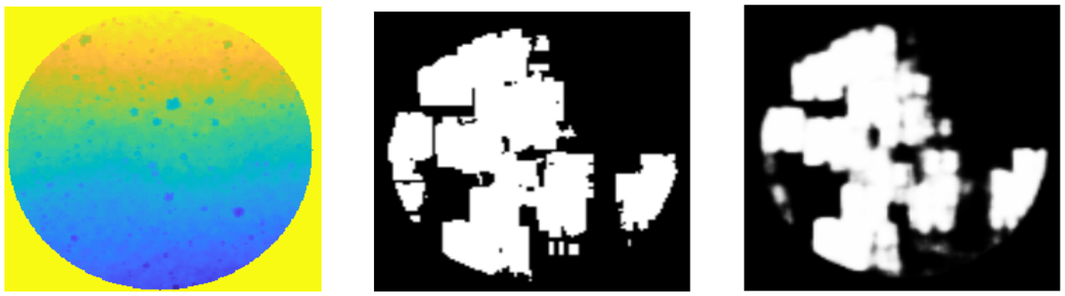
In order to find a more accurate heuristic for ACE cost, we trained a model to predict ACE values based on heightmap data (Figure 6.1). In contrast to the Gradient Convolution heuristic, which was hand-coded by domain experts, the learned heuristic is automatically encoded using a data-driven framework. More specifically, we developed a deep convolutional neural network (DCNN) based model that can directly predict the outcome of the ACE algorithm for a given terrain heightmap. Using this prediction, ENav can more optimally sort its initial list of potential paths and hence reduce the average number of ACE evaluations required until finding a safe path.

We formulate this problem as a supervised-learning based classification. Our DCNN model is based on a modified encoder-decoder style U-Net architecture Ronneberger, Fischer, and Brox, 2015. The encoder consists of a series of convolutional layers that down-samples the input to a low-dimensional feature map, and a decoder that consists of up-sampling layers with convolutions that then take this feature map and increase their resolution to that of the original input. U-Net also has a series of residual connections from the encoder to the decoder feature maps that helps restore the high-resolution details lost during down-sampling and also prevents vanishing gradients during training.

The input to our model is a heightmap and the output is an ACE map, such that the value for each pixel in the ACE map corresponds to the expected ACE cost for the corresponding terrain parameters. However, ACE cost depends not only on the terrain but also on the rover heading. We encode the rover heading as part of the learning problem itself by extending the output to have a multi-channel representation such that each channel represents a cardinal heading angle for the rover. In our experiments, we have found a discretization of 8 heading angles (at 45 degree intervals) to be sufficient. Sigmoid activation is applied to each channel to give a value in the range [0, 1] corresponding to the probability of a cell being infinite ACE cost or not.

Training data was gathered by running a Monte Carlo simulation of the baseline ENav algorithm on 1500 terrains, randomly sampling 8 heightmaps from each trial. For each cell in each sampled heightmap, the ACE algorithm was run with the eight fixed rover heading values, resulting in an "ACEmap" where each cell has eight heading-specific values. Of the 12000 total heightmap, ACE map pairs, 9500 were used as a training set, and 2500 were used as the validation set. The learned heuristic model achieved 97.8% training accuracy and 95.3% validation accuracy.

Note the result of this prediction is a probability of ACE returning a safety violation, which is different from the output of the ACE algorithm itself. ACE can return finite or infinite costs, where finite costs represent how close the rover is to safety violations, and infinite costs represent safety violations. Predicting the actual ACE costs is a dual problem of classification and regression, and is more difficult that segmentation alone. Efforts to predict these values have not yet yielded results.



(a) Height Map (b) ACE Map (c) Inferred ACE Map

Figure 6.1: An example of a Learned Heuristic. Sets of terrain heightmaps (a) and maps generated by the ACE algorithm (b) were used to train a neural network to generate an inferred ACE probability map (c).

6.3 Evaluations of the Learned Safety Surrogate

Described at length in Toupet et al., 2020, a Monte Carlo simulation environment was built for testing ENav and Mars 2020 navigation. The Robotics Operating System (ROS) Quigley et al., 2009 was used for inter-process communication. One ROS node wraps ENav, and another wraps the HyperDrive Simulator (HDSim), which simulates rover motion, terrain settling and slipping, and disparity images for JPL rover missions. Simulated terrains, representing various slopes and rock densities, are loaded into HDSim. Rock densities are classified by the cumulative fractional area covered by rocks (CFA) Golombek and Rapp, 1997.

The rover starts at one side of the map and ENav is given a global goal 80m away at the other end of the map. The trial is run until the rover either successfully reaches the goal, or until a failure condition is found, such as when no feasible path can be found, when the safety limits of the rover are violated, or when the duration of the trial exceeds a time limit.

We leverage this existing simulation setup to run our experiments. In the case of simulations employing the machine learning model, we use an additional ROS node running TensorFlow, which receives heightmaps and publishes ACE estimates. Because HDSim currently only runs on 32-bit systems, and TensorFlow only runs on 64-bit systems, we use a ROS multi-master system to communicate between two computers.

The discussion of the experiments notes potential gains in computation time. These gains are theoretical and predicated on the notion that each call of the ACE algorithm takes 10 to 20 ms on the RAD750 flight processor, these calls make up a very large portion of ENav's computation time, and reduction of these calls translates to a gain in computation time. The use of non-flight-like computers, the need for multi-master ROS, and lack of optimization in the added algorithms for this research prevent meaningful comparisons of wall clock time.

The results of the following experiments are summarized in Figure 6.2. Each simulation tracks results on two subsets of terrains: **Benign** terrains have a CFA value of 7% or less, and a slope of 15° or less. **Complex** terrains have greater slope or CFA. Each performance metric is calculated as a weighted average across each subset of terrains (Benign or Complex), with terrains of greater complexity being less likely to occur on Mars, and accordingly given less weight. We are more concerned with tracking performance on complex terrain. Within each Monte Carlo simulation, 780 trials are run on complex terrain, so $n = 780$ will be used when

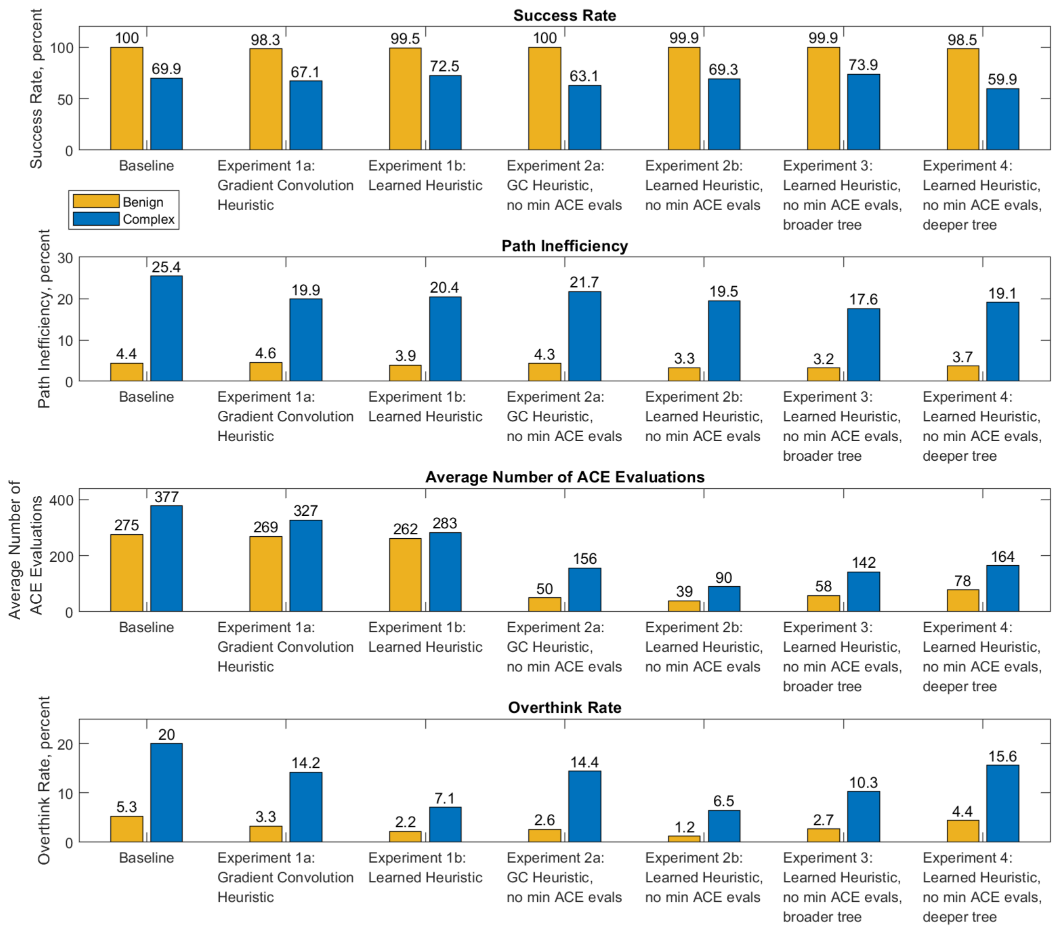


Figure 6.2: A summary of key rover path planning performance metrics across various experiments. Incorporating heuristics produced more efficient paths, reduced the number of costly ACE evaluations, and maintained or slightly increased the rate of successfully reaching the goal.

calculating 95% confidence margins of error (MOE).

The following performance metrics are tracked:

- **Success Rate** is the percentage of trials for each terrain that result in the rover reaching the global goal without timing out, reaching a point with no feasible paths, or violating safety constraints. Higher values are better.

- **Average Path Inefficiency** is defined as the average length of the path taken by the rover divided by the Euclidean distance from the start to the goal, minus 1, expressed as a percentage. For example, if the rover goal was 100 m away, and the rover needed to travel a circuitous route with a length of 125 m to avoid obstacles and reach the goal, the path inefficiency was 25%. Lower values are better.

- **Average ACE Evaluations** is the average number of ACE evaluations conducted per planning cycle. Lower values are better. Each evaluation takes an estimated 10 to 20 ms on a RAD750 processor like that on Perseverance, so a conversion to average cycle time can easily be made.

- **Overthink Rate** is the average percent of ENav planning cycles that required more ACE evaluations than a threshold value (275 by default), which indicates that the highest-ranked candidate paths were all deemed unsafe by ACE, and therefore the initial ranking of paths was unsuitable. When the number of ACE evaluations exceeds the threshold, it indicates that ENav is "overthinking" and the rover may need to stop driving until a solution is found. Lower values are better.

Our main results are shown in Figure 6.2. The simulation with the learned heuristic showed improvements across almost every metric (Figure 6.3, Experiment 1b). Complex success rate improved slightly, from 69.9% to 72.5% (within the MOE). Path inefficiency significantly improved, especially for complex terrains, going from 25.4% to 20.4%. The number of ACE evaluations also reduced, especially in terms of the overthink rate, which plummeted to 7.1% compared to 20.0% for the baseline. This would result in far fewer cases of the rover needing to stop driving before the next path can be found.

We asserted that ML could be added without sacrificing the safety guarantee of the ACE algorithm, and this assertion holds. No trial failures were caused by violated safety constraints; all failures are due to either timeouts or failures to find paths to the goal. This was true for all experiments.

This experiment answers the most fundamental question of this work: can heuristics, designed or learned, improve the performance of the baseline rover navigation algorithm? The answer is yes. Heuristics can more effectively rank the set of candidate paths, reduce the average computation time needed to find a safe path, choose maneuvers that increase path efficiency, and increase the likelihood of successfully reaching the goal, while maintaining rover safety.

6.4 Learning Surrogate with Submodular-Norm Regularization

In real-world automated decision making tasks we seek the optimal set of actions that jointly achieve the maximal utility. Many of such tasks, either deterministic/non-adaptive or stochastic/adaptive, can be viewed as combinatorial optimization problems over a large number of actions. As an example, consider the active learning problem where a learner seeks the maximally-informative set of training examples for learning a classifier. The utility of a training set could be measured by the mutual information (Lindley, 1956) between the training set and the remaining (unlabeled) data points, or by the expected reduction in generation error if the model is trained on the candidate training set. Similar problems arise in a number of other domains, such as experimental design (Chaloner and Verdinelli, 1995), document summarization (Lin and Bilmes, 2012), recommender system (Javdani et al., 2014), and policy making (Runge, Converse, and Lyons, 2011).

For a broad class of decision making problems whose optimization criterion is to maximize the decision-theoretic *value of information* (e.g., active learning and experimental design), it has been shown that it is possible to design surrogate objective functions that are (approximately) submodular while being aligned with the original objective at the optimal solutions (Javdani et al., 2014; Y. Chen, Javdani, et al., 2015; Choudhury et al., 2017). Despite the promising performance, a caveat for these "submodular surrogate"-based approaches is that it is often challenging to engineer such a surrogate objective without an ad-hoc design and analysis that requires trial-and-error (Y. Chen, Javdani, et al., 2015; Satsangi et al., 2018). Furthermore, for certain classes of surrogate functions, it is NP-hard to compute/evaluate the function value (Javdani et al., 2014). In such cases, even a greedy policy, which iteratively picks the best action given the (observed) history, can be prohibitively costly to design or run. Addressing this limitation requires more automated or systematic ways of designing (efficient) surrogate objective functions for decision making.

Inspired by contemporary work in data-driven decision making, we aim to learn a greedy heuristic for sequentially selecting actions. This heuristic acts as a surrogate for invoking the expensive oracle when evaluating an action. Our key insight is that many practical algorithms can be interpreted as greedy approaches that follow an (approximate) submodular surrogate objective. In particular, we focus on the class of combinatorial problems that can be solved via submodular maximization (either directly on the objective function or via a submodular surrogate). We highlight some of the key results below:

- Focusing on utility-based greedy policies, we introduce a data-driven optimization framework based on the "*submodular-norm*" loss, which is a novel loss function that encourages learning functions that exhibit "diminishing returns". Our framework, called LEASURE (Learning with Submodular Regularization), outputs a surrogate objective that is efficient to train, approximately submodular, and can be made permutation-invariant. The latter two properties allow us to prove approximation guarantees for the resulting greedy heuristic.

- We show that our approach can be easily integrated with modern imitation learning pipelines for sequential prediction tasks. We provide a rigorous analysis of the proposed algorithm and prove strong performance guarantees for the learned objective.

- We demonstrate the performance of our approach on a data-driven protein design task. Our results suggest that, compared to standard learning-based baselines: (a) at training time, LEASURE requires significantly fewer oracle calls to learn the target objective (i.e., to minimize the approximation error against the oracle objective); and (b) at test time, LEASURE achieves superior performance on the corresponding optimization task (i.e., to minimize the regret for the original combinatorial optimization task). In particular, LEASURE has shown promising performance in the protein design task and will be incorporated into a real-world protein design workflow.

6.5 Background and Problem Statement

Decision Making via Submodular Surrogates

Given a ground set of items $\mathcal{V}$ to pick from, let $u : 2^{\mathcal{V}} \to \mathbb{R}$ be a set function that measures the *value* of any given subset[1] $A \subseteq \mathcal{V}$. For example, for experimental design, $u(A)$ captures the utility of the output of the best experiment; for active learning $u(A)$ captures the generalization error after training with set A. We denote a policy $\pi : 2^{\mathcal{V}} \to \mathcal{V}$ to be a partial mapping from the set/sequence of items already selected, to the next item to be picked. We use Π to denote our policy class. Each time a policy picks an item $e \in \mathcal{V}$, it incurs a unit cost. Given the ground set $\mathcal{V}$, the utility function u, and a budget k for selecting items, we seek the optimal policy π

[1]For simplicity, we focus on deterministic set functions in this section. Note that many of our results can easily extent to the stochastic, by leveraging the theory of adaptive submodularity (Golovin and Krause, 2011)

that achieves the maximal utility:

$$\pi^* \in \arg\max_{\pi \in \Pi} u(S_{\pi,k}). \tag{6.1}$$

$S_{\pi,k}$ is the sequence of items picked by π: $S_{\pi,i} = S_{\pi,i-1} \cup \{\pi(S_{\pi,i-1})\}$ for $i > 0$ and $S_{\pi,0} = \emptyset$.

As we have discussed in the previous sections, many sequential decision making problems can be characterized as constrained monotone submodular maximization problem. In those scenarios u is:

- **Monotone**: For any $A \subseteq \mathcal{V}$ and $e \in \mathcal{V} \setminus A$, $u(A) \leq u(A \cup \{e\})$.
- **Submodular**: For any $A \subseteq B \subseteq \mathcal{V}$ and $e \in \mathcal{V} \setminus B$, $u(A \cup \{e\}) - u(A) \geq u(B \cup \{e\}) - u(B)$.

In such cases, a mypopic algorithm following the greedy trajectory of u admits a near-optimal policy. However, in many real-world applications, u is not monotone submodular. Then one strategy is to design a surrogate function $f : 2^{\mathcal{V}} \to \mathbb{R}$ which is:

- Globally aligning with u: For instance, f lies within a factor of u: $f(A) \in [c_1 \cdot u(A), c_2 \cdot u(A))]$ for some constants c_1, c_2 and any set $A \subseteq \mathcal{V}$; or within a small margin with u: $f(A) \in [u(A) - \epsilon, u(A) + \epsilon]$ for a fixed $\epsilon > 0$ and any set $A \subseteq \mathcal{V}$;
- Monotone submodular: Intuitively, a submodular surrogate function encourages selecting items that are beneficial in the long run, while ensuring that the decision maker does not miss out any actions that are "surprisingly good" by following a myopic policy (i.e., future gains for any item are diminishing). Examples that fall into this category include machine teaching (Singla et al., 2014), active learning (Y. Chen, Hassani, et al., 2015), etc.

We argue that in real-world decision making scenarios—as validated later in Section 6.8—the decision maker is following a surrogate objective that aligns with the above characterization. In the following context, we will assume that such surrogate function exists. Our goal is thus to learn from an *expert policy* that behaves greedily according to such surrogate functions.

Learning to Make Decisions

We focus on the regime where the expert policy is expensive to evaluate. Let $g : 2^{\mathcal{V}} \times \mathcal{V} \to \mathbb{R}$ be the score function that quantifies the benefit of adding a new item to an existing subset of $\mathcal{V}$. For the expert policy and submodular surrogate f discussed in Section 6.5, $\forall A \subseteq \mathcal{V}$ and $e \in \mathcal{V}$:

$$g^{\text{exp}}(A, e) = f(A \cup \{e\}) - f(A).$$

For example, in the active learning case, $g^{\text{exp}}(A, e)$ could be the expert acquisition function that ranks the importance of labelling each unlabelled point, given the currently labelled subset. In the set cover case, $g^{\text{exp}}(A, e)$ could be the function that gives the score to each vertex and determines the next best vertex to add to the cover set. Given a loss function ℓ, our goal is to learn a score function $\hat{g}$ that incurs the minimal expected loss when evaluated against the expert policy: $\hat{g} = \arg\min_g \mathbb{E}_{A,e}[\ell(g(A, e), g^{\text{exp}}(A, e))]$. Subsequently, the utility by the learned policy is $u(S_{\hat{\pi},k})$, where for any given history $A \subseteq \mathcal{V}$, $\hat{\pi}(A) \in \arg\max_{e \in \mathcal{V}} \hat{g}(A, e)$.

6.6 Learning with Submodular Regularization

To capture our intuition that a greedy expert policy tends to choose the most useful items, we introduce LeaSuRe, a novel regularizer that encourages the learned score function (and hence surrogate objective) to be submodular. We describe the algorithm below.

Given the groundset $\mathcal{V}$, let $f : 2^{\mathcal{V}} \to \mathbb{R}$ be any approximately submodular surrogate such that $f(A)$ captures the "usefulness" of the set A. The goal of a trained policy is to learn a score function $g : 2^{\mathcal{V}} \times \mathcal{V} \to \mathbb{R}$ that mimics $g^{\text{exp}}(A, x) = f(A \cup \{x\}) - f(A)$, which is often prohibitively expensive to evaluate exactly. Then, given any such g, we can define a greedy policy $\pi(A) = \text{argmax}_{x \in \mathcal{V}} g(A, x)$. With LeaSuRe, we aim to learn such function g that approximates g^{exp} well while being inexpensive to evaluate at test time. Let $D_{real} = \{(\langle A, x\rangle, y^{\text{exp}} = g^{\text{exp}}(A, x))\}_m$ be the gathered tuple of expert scores for each set-element pair. If the set $2^{\mathcal{V}} \times \mathcal{V}$ was not too large, the LeaSuRe could be trained on the randomly collected tuples D_{real}. However, $2^{\mathcal{V}}$ tends to be too large to explore, and generating ground truth labels could be very expensive. To leverage that, for a subset of set-element pairs in D_{real} we generate a set of random supersets to form an unsupervised synthetic dataset of tuples $D_{synth} = \{(\langle A, x\rangle, \langle A', x\rangle) | A \preceq A', \langle A, x\rangle \in D_{real}\}_n$ where A'

denote a randomly selected superset of A. Define:

$$\begin{aligned}\text{Loss}(g, g^{\text{exp}}) = &\sum_{\langle A,x\rangle, y^{\text{exp}} \in D_{real}} (y^{\text{exp}} - g(A,x))^2 + \\ &\lambda \sum_{(\langle A,x\rangle, \langle A',x\rangle) \in D_{synth}} \sigma([g(A',x) - g(A,x)]).\end{aligned}$$

where $\lambda > 0$ is the regularization parameter and σ is the sigmoid function. Intuitively, such regularization term will force the learned function g to be close to submodular, as it will lead to larger losses every time $g(A',x) > g(A,x)$. If we expect f to be monotonic, we also introduce a second regularizer ReLu$(-g(A',x))$ which pushes the learned function to be positive. Combined, the loss function becomes (used in Line 11 in Algorithm 13):

$$\begin{aligned}\text{Loss}(g, g^{\text{exp}}) = &\sum_{\langle A,x\rangle, y^{\text{exp}} \in D_{real}} (y^{\text{exp}} - g(A,x))^2 \\ &+ \lambda \sum_{(\langle A,x\rangle, \langle A',x\rangle) \in D_{synth}} \sigma([g(A',x) - g(A,x)]) \\ &+ \gamma \sum_{\langle A',x\rangle \in D_{synth}} \text{ReLu}(-g(A',x)),\end{aligned}$$

where γ is another regularization strength parameter. Such loss should push g to explore a set of approximately submodular, approximately monotonic functions. Thus, if f exhibits the submodular and monotonic behavior, g trained on this loss function should achieve a good local minima.

We next note that since $2^{\mathcal{V}}$ is too large to explore, instead of sampling random tuples for D_{real}, we use modified DAgger. Then g can learn not only from the expert selections of $\langle A, x\rangle$, but it can also see the labels of the tuples the expert would not have chosen.

Algorithm 13 above describes our approach. A trajectory in Line 7 is a sequence of iteratively chosen tuples, $(\langle\emptyset, x_1\rangle, \langle\{x_1\}, x_2\rangle, \langle\{x_1, x_2\}, x_3\rangle ..., \langle\{x_1, ..., x_{T-1}\}, x_T\rangle)$, collected using a mixed policy π_i. In Line 8, expert feedback of selected actions is collected to form D_i. Note that in some settings, even collecting exact expert labels g^{exp} at train time could be too expensive. In that case, g^{exp} can be replaced with a less expensive, noisy approximate expert $g^{\text{exp}}_{\epsilon} \approx g^{\text{exp}}$. In fact, all three of our experiments use noisy experts in one form or another.

6.7 Analysis for LeaSuRe

Estimating the expert's policy. We first consider the bound on the loss of the learned policy measured against the expert's policy. Since LeaSuRe can be viewed

Algorithm 13 Learning to make decisions via Submodular Regularization (LeaSuRe)

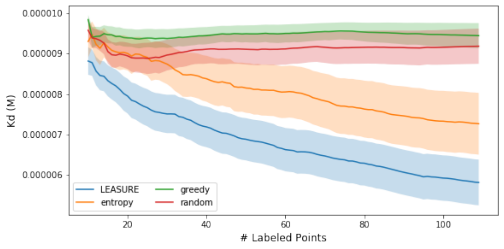
```
1: Input: Ground set $\mathcal{V}$, expert score function $g^{\exp}$,
   regularization parameters $\lambda, \gamma$, DAgger constant $\beta$, the length of trajectories $T$.

2: initialize $D_{real} \leftarrow \emptyset$
3: initialize $g$ to any function.
4: for $i = 1$ to $N$ do
5:     Let $g_i = g^{\exp}$ with probability $\beta$.
6:     Sample a batch of $T-$step trajectories using $\pi_i(A) = x_i = \text{argmax}_{x \in \mathcal{V}} g_i(A, x)$.
7:     Get dataset $D_i = \{\langle A_i, x_i \rangle, g^{\exp}(A_i, x_i)\}$ of labeled tuples on actions taken by $\pi_i$.
8:     $D_{real} \leftarrow D_{real} \bigcup D_i$.
9:     Generate synthetic dataset $D_{synth}$ from $D_{real}$.
10:    Train $g_{i+1}$ on $D_{real}$ and $D_{synth}$ using the loss function above.
11: end for
12: Output: $g_{N+1}$
```

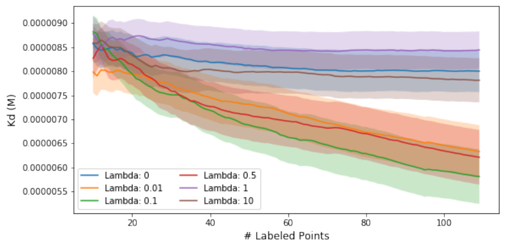
as a specialization of DAgger (Stéphane Ross, Gordon, and D. Bagnell, 2011) for learning a submodular function, it naturally inherits the performance guarantees from DAgger, which show that the learned policy efficiently converges to the expert's policy. Concretely, the following result, which is adapted from the original DAgger analysis, shows that the learned policy is consistent with the expert policy and thus is a *no-regret* algorithm:

Theorem 8 (Theorem 3.3, Stéphane Ross, Gordon, and D. Bagnell, 2011)**.** *Denote the loss of $\hat{\pi}$ at history state H as $l(H, \hat{\pi}) := \ell(g(H, \hat{\pi}(H)), g^{exp}(H, \pi^{exp}(H)))$. Let $d_{\hat{\pi}}$ be the average distribution of states if we follow $\hat{\pi}$ for a finite number of steps. Furthermore, let D_i be a set of m random trajectories sampled with π_i at round $i \in \{1, \ldots, N\}$, and $\hat{\epsilon}_N = \min_\pi \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{H_i \sim D_i}[l(H_i, \hat{\pi})]$ be the training loss of the best policy on the sampled trajectories. If N is $\mathcal{O}(T^2 \log(1/\delta))$ and m is $\mathcal{O}(1)$ then with probability at least $1 - \delta$ there exists a $\hat{\pi}$ among the N policies, with $\mathbb{E}_{H \sim d_{\hat{\pi}}}[l(H, \hat{\pi})] \le \hat{\epsilon}_N + \mathcal{O}\left(\frac{1}{T}\right)$.*

Approximating the optimal policy. Note that the previous notion of regret corresponds to the average difference in score function between the learned policy and the expert policy. While this result shows that LeaSuRe is consistent with the expert, it does not directly address how well the learned policy performs in terms of the gained utility. We then provide a bound on the expected value of the learned



(a) Comparison to baseline methods



(b) Effect of scaling parameter lambda

Figure 6.3: Combining submodular regularization with a learned active learning policy for a protein engineering task. In (b), Lambda = 0 corresponds to the unregularized case. Error bars are plotted as standard error of the mean across 50 replicates.

policy, measured against the value of the optimal policy. For specific decision making tasks where the oracle follows an approximately submodular objective, our next result, which is proved in the appendix, shows that the learned policy behaves near-optimally.

Theorem 9. *Assume that the utility function u is monotone submodular. Furthermore, assume the expert policy π^{exp} follows a surrogate objective f such that for all $A \subseteq \mathcal{V}$, $|f(A) - u(A)| < \epsilon_E$ where $\epsilon_E > 0$. Let $\hat{\epsilon}_N = \min_\pi \frac{1}{N} \sum_{i=1}^{N} l(H_i, \hat{\pi})$ be the training loss of the best policy on the sampled trajectories. If N is $\mathcal{O}\left(T^2 \log(1/\delta)\right)$ then with probability at least $1 - \delta$, the expected utility achieved by running $\hat{\pi}$ for k steps is*

$$\mathbb{E}[u(S_{\hat{\pi},k})] \geq (1 - 1/e)\mathbb{E}[u(S_{\pi^*,k})] - k(\epsilon_E + \Delta_{\max}\hat{\epsilon}_N) - O(1).$$

A closely related work in approximate policy learning is by Stephane Ross, Zhou, et al., 2013, which also builds upon DAGGER to tackle policy learning for submodular optimization, via directly imitating the greedy oracle decision rather than learning a surrogate utility. One key difference is that their approach can only yield guarantees against an artificial benchmark (a set or list of simpler policies that each independently selects an item to add to the action set), whereas our theoretical guarantees are with respect to the optimal policy in our class.

6.8 Evaluations LeaSuRe on Protein Engineering

In this section, we demostrate the performance of LeaSuRe on a protein engineering task.

Protein Engineering

By employing a large protein engineering database containing mutation-function data (C. Y. Wang et al., 2019), we demonstrate that LeaSuRe enables the learning of an optimal policy for imitating expert design of protein sequences (see Appendix for detailed discussion of datasets). As in Liu, Buntine, and Haffari (2018) we construct a fully data-driven "expert" which evaluates via 1-step roll-out the effect of labeling each candidate data (in our case a protein mutant) with the objective of minimizing loss on a downstream regression task (predicting protein fitness).

When training the policy to emulate the algorithmic expert via imitation learning, we represent each state as two merged representations: (1) a fixed dimensional representation of the protein being considered (as the last dense layer of the network described in Appendix C), and (2) a similar fixed dimensional representation of the data already included in the training set (as a sum of their embeddings), including their average label value. At each step a random pool of data is drawn from the state space and the expert policy greedily selects a protein to label, which minimizes the expected regression loss on the downstream regression task (prediction of protein fitness). Once the complete pool of data has been evaluated, the states are stored along with their associated preference score, taken as their ability to reduce the loss in the 1-step roll out. Using these scores, the expert selects a protein sequence to add into the training set, and we retrain the model and use the updated model to predict a protein with the maximum fitness. This paired state action data is used to train the policy model at the end of each episode, as described in Liu, Buntine, and Haffari (2018). As we observe in Figure 6.3a, this method trains a policy which performs nearly identically to this 1-step oracle expert.

The use of submodular regularization enables the learning of a policy which generalizes to a fundamentally different protein engineering task. In our experiments, LeaSuRe is trained to emulate a greedy oracle for maximizing the stability of protein G, a small bacterial protein used across a range of biotechnology applications (Sjöbring, Björck, and Kastern, 1991). We evaluate our results by applying the trained policy to select data for the task of predicting antibody binding to a small molecule. As is the case with all protein fitness landscapes, the evaluation dataset is highly imbalanced, with the vast majority of mutants conferring no improvement at all. Because data is expensive to label in biological settings (proteins must be synthesized, purified and tested), we are often limited in how many labels can feasibly be generated, and the discriminative power among the best results is often more important than among the worst. To construct a metric with real-world applicability we assess each model by systemically examining the median Kd of the next ten data points selected at each budget, from 10 to 110 total labels. This method is utilized in recognisance of the extreme ruggedness of protein engineering landscapes, wherein the vast majority of labels are of null fitness, and the ability to select rare useful labels for the next experimental cycle is of key importance.

We observe that LEASURE outperforms all evaluated baselines, and that the inclusion of submodular optimization is mandatory to its success (Figure 6.3a). A greedy active learner which labels the antibody mutation with the best predicted Kd (the smallest) preforms approximately equivalently with selecting random labels. Use of dropout as an approximation of model uncertainty as in Gal and Ghahramani (2016) improves upon these baselines, although significant betterment is not achieved until approximately 35 labels are added. In comparison, the results from LEASURE diverge from all others nearly immediately, and the best model, which uses a lambda of 0.1, achieves a notable improvement in Kd, 5.81μM, vs 7.27μM achieved by entropy sampling. In support of methods success, we note that the learned policy preforms approximately as well as the greedy oracle which it emulates (Appendix Figure C.1a). We observe that the results are robust within a range of possible lambda values (Figure Figure 6.3b and Appendix Figure C.1b), and that without the use of submodular regularization the trained policy fails to learn a policy better than the selection of random labels. This is an important finding, as the method proposed by Liu, Buntine, and Haffari (2018) without LEASURE, has been shown to be a state-of-the-art method for imitation learning.

Based on these empirical results, LEASURE demonstrates significant potential as computational tool for *real-world automated experimental design tasks*: In particular, in the protein engineering task, LEASURE achieves the SOTA on the benchmark data-sets considered in this work. While LEASURE does involve repeated retraining of the protein engineering network, we observe that it returns strong results even with a single step of training. Additionally, the networks that are employed are very simple (Appendix C). This allows for reasonable training time (36 hours) and nearly instantaneous inference. Given the considerable time and cost of protein engineering, these computational budgets are quite modest. Protein engineering is a time consuming (months to years) and expensive undertaking (10's of thousands to millions of dollars). These projects usually strive to achieve the best possible results given a fixed budget. We have demonstrated in our work the ability deliver significant improvements in protein potency for the modest fixed budgets. Although the cost savings of engineering and testing an individual protein (or label) vary significantly based on the system, ranging tens to hundreds of dollars, we observe that to achieve a Kd of 8e-6 M LeaSuRe delivers an approximate cost savings of 65%, or 40 fewer labels than the next best method. The sequential synthesis and evaluation of each of these labels would likely span several months and additionally incur several thousands of dollars of materials costs.

Chapter 7

CONCLUSION & FUTURE DIRECTIONS

In this thesis, we have presented several approaches to incorporate learning into optimization algorithms. We covered methods from replacing a single component of a solver to incorporating a complete solver as a sub-routine. Both theoretical analyses and practical applications are considered. Many interesting research questions remain to be explored and we describe several below.

Learning Interacting Components In chapters 3 and 4, we presented the retrospective imitation learning and co-training policy learning algorithms and applied them to learn the node selection module in the branch-and-bound algorithm. Other works have shown successes in learning the branching module and the cutting-plane selection module. A practical question is how to combine different data-driven components so that they work well together. As these modules interact closely, e.g., the branching decisions determine the candidates for node selection, we may want to design learning algorithms that consider those interactions.

In chapter 4, we showed how to incorporate learning with the large neighborhood search framework and with the backdoor prediction idea. Both methods use a solver as a sub-routine. In principle, an improved solver will lead to better performance since both methods rely on an underlying solver. Thus it is worth considering how the component improvements can be used jointly with those high-level learning methods. We can envision a feature-rich learning-based solver that utilizes machine learning models in every level of decision-making.

Differentiating through Solvers In this thesis, we focused on formulating optimization as sequential decision-making problems. The benefit is that we can connect with existing research in policy learning and adapt them for specific problem settings. A challenge with policy learning algorithms is their demand for large amounts of environmental interactions. Both the online approaches, iteratively collecting data during learning (chapters 3, 4 and 6), and the offline approaches, collecting data before learning (chapters 5 and 6), have the same bottleneck. As a potential remedy, end-to-end differentiable approaches hold promising answers. With the recent appearances of such methods, we can explore corresponding versions in the learning tasks we have considered.

Interpreting the Learned Components Can we explain the behavior of a learned decision module? Is it possible to distill a machine learning model to a mathematical formula so that domain experts can analyze its theoretical property? We believe understanding these questions can benefit the learning to optimize community immensely by bringing more solid theoretical groundings. By inspecting the distilled version, we can potentially discover unknown structures about a particular optimization problem. As a result, the learning community can contribute to the optimization community by using models as a discovery tool.

Expanding to Other Applications Together with concrete algorithm development, we also presented general perspectives on learning to optimize, such as divide-and-conquer to incorporate solvers. We believe there are many more applications that can benefit from such perspectives. Some applications with combinatorial search space include symbolic reasoning such as theorem proving and physical design problems in hardware. We see similar patterns in these areas: domain-specific heuristics crafted by hand are prevalent and historical data are available. Thus, they should also be amenable to data-driven approaches.

BIBLIOGRAPHY

Abbeel, Pieter and Andrew Y Ng (2004). “Apprenticeship learning via inverse reinforcement learning”. In: *International Conference on Machine Learning*.

Abcouwer, Neil et al. (2021). “Machine Learning Based Path Planning for Improved Rover Navigation”. In: *IEEE Aerospece Conference*. URL: `https://arxiv.org/abs/2011.06022`.

Achterberg, Tobias (2009). “SCIP: solving constraint integer programs”. In: *Mathematical Programming Computation* 1.1, pp. 1–41.

Achterberg, Tobias and Roland Wunderling (2013). “Mixed integer programming: Analyzing 12 years of progress”. In: *Facets of combinatorial optimization*. Springer, pp. 449–481.

Ahuja, Ravindra K et al. (2002). “A survey of very large-scale neighborhood search techniques”. In: *Discrete Applied Mathematics* 123.1-3, pp. 75–102.

Albert, Réka and Albert-László Barabási (2002). “Statistical mechanics of complex networks”. In: *Reviews of modern physics* 74.1, p. 47.

Alieva, Ayya, Aiden Aceves, Jialin Song, Stephen Mayo, Yisong Yue, and Yuxin Chen (2021). “Learning to Make Decisions via Submodular Regularization”. In: *International Conference on Learning Representations*. URL: `https://openreview.net/forum?id=ac288vnG_7U`.

Alley, E. C. et al. (2019). “Unified rational protein engineering with sequence-based deep representation learning”. In: *Nat. Methods*.

Amos, Brandon and J Zico Kolter (2017). “Optnet: Differentiable optimization as a layer in neural networks”. In: *International Conference on Machine Learning*. PMLR, pp. 136–145.

Andrychowicz, Marcin et al. (2016). “Learning to learn by gradient descent by gradient descent”. In: *Advances in neural information processing systems*, pp. 3981–3989.

Ansótegui, Carlos et al. (2015). “Model-based genetic algorithms for algorithm configuration”. In: *AISTATS*.

Aytar, Yusuf et al. (2018). “Playing hard exploration games by watching YouTube”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 2935–2945.

Balcan, Maria-Florina et al. (2018). “Learning to Branch”. In: *ICML*.

Balunovic, Mislav, Pavol Bielik, and Martin Vechev (2018). “Learning to Solve SMT Formulas”. In: *Neural Information Processing Systems (NeurIPS)*.

Bar-Yehuda, Reuven and Shimon Even (1983). *A Local-Ratio Theorm for Approximating the Weighted Vertex Cover Problem*. Tech. rep. Computer Science Department, Technion.

Bérard, Béatrice et al. (2013). *Systems and software verification: model-checking techniques and tools*.

Berthold, Timo (2006). "Primal heuristics for mixed integer programs". In:

Bliek1ú, Christian, Pierre Bonami, and Andrea Lodi (2014). "Solving mixed-integer quadratic programming problems with IBM-CPLEX: a progress report". In: *RAMP*.

Blum, Avrim and Tom Mitchell (1998). "Combining labeled and unlabeled data with co-training". In: *Conference on Learning Theory (COLT)*.

Bojarski, Mariusz et al. (2016). "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316*.

Boros, Endre and Peter L Hammer (1991). "The max-cut problem and quadratic 0–1 optimization; polyhedral aspects, relaxations and bounds". In: *Annals of Operations Research*.

Boyd, Stephen and Lieven Vandenberghe (2004). *Convex optimization*. Cambridge university press.

Brockman, Greg et al. (2016). "Openai gym". In: *arXiv preprint arXiv:1606.01540*.

Burer, Samuel, Renato DC Monteiro, and Yin Zhang (2002). "Rank-two relaxation heuristics for max-cut and other binary quadratic programs". In: *SIAM Journal on Optimization* 12.2, pp. 503–521.

Burges, Chris, Erin Renshaw, and Matt Deeds (1998). "Learning to Rank using Gradient Descent". In: *International conference on Machine learning*.

Burges, Chris, Tal Shaked, et al. (2005). "Learning to rank using gradient descent". In: *International Conference on Machine Learning (ICML)*.

Chaloner, K. and I. Verdinelli (1995). "Bayesian experimental design: A review". In: *Statistical Science* 10.3, pp. 273–304.

Chang, Kai-Wei et al. (2015). "Learning to search better than your teacher". In: *International Conference on Machine Learning*. PMLR, pp. 2058–2066.

Chen, Binghong, Bo Dai, and Le Song (2020). "Learning to Plan via Neural Exploration-Exploitation Trees". In: *ICLR*.

Chen, Yuxin, S Hamed Hassani, et al. (2015). "Sequential information maximization: When is greedy near-optimal?" In: *Conference on Learning Theory*, pp. 338–363.

Chen, Yuxin, Shervin Javdani, et al. (Jan. 2015). "Submodular Surrogates for Value of Information". In: *Proc. Conference on Artificial Intelligence (AAAI)*.

Chen, Yuxin, Jean-Michel Renders, et al. (2017). “Efficient Online Learning for Optimizing Value of Information: Theory and Application to Interactive Troubleshooting”. In: *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI 2017)*. Vol. 2. Curran Associates, Inc., pp. 966–983.

Cheng, Ching-An et al. (2018). “Fast policy learning through imitation and reinforcement”. In: *Conference on Uncertainty in Artificial Intelligence*.

Choudhury, Sanjiban et al. (2017). “Learning to gather information via imitation”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 908–915.

Colombi, Marco, Renata Mansini, and Martin Savelsbergh (2017). “The generalized independent set problem: Polyhedral analysis and solution approaches”. In: *European Journal of Operational Research* 260.1, pp. 41–55.

Conforti, Michele, Gérard Cornuéjols, Giacomo Zambelli, et al. (2014). *Integer programming*. Vol. 271. Springer.

Cormen, Thomas H et al. (2009). *Introduction to algorithms*. MIT press.

Cover, Thomas M and Joy A Thomas (2012). *Elements of information theory*. John Wiley & Sons.

Cui, Jun et al. (2006). “Combinatorial search of thermoelastic shape-memory alloys with extremely small hysteresis width”. In: *Nature materials*.

Dai, Hanjun, Bo Dai, and Le Song (2016). “Discriminative Embeddings of Latent Variable Models for Structured Data”. In: *International Conference on Machine Learning*, pp. 1–23.

Dai, Hanjun, Elias B Khalil, et al. (2017). “Learning combinatorial optimization algorithms over graphs”. In: *Neural Information Processing Systems*.

Dasgupta, Sanjoy, Michael L Littman, and David A McAllester (2002). “PAC generalization bounds for co-training”. In: *Neural information processing systems*.

Daumé III, Hal, John Langford, and Daniel Marcu (2009). “Search-based structured prediction”. In: *Machine learning* 75.3, pp. 297–325.

De Jong, Kenneth (Jan. 2006). *Evolutionary Computation – A Unified Approach*. ISBN: 978-0-262-04194-2.

De Moura, Leonardo and Nikolaj Bjørner (2008). “Z3: An efficient SMT solver”. In: *TACAS*.

De Vries, Sven and Rakesh V Vohra (2003). “Combinatorial auctions: A survey”. In: *INFORMS Journal on computing*.

Dechter, Rina and Judea Pearl (1985). “Generalized best-first search strategies and the optimality of A”. In: *Journal of the ACM (JACM)* 32.3, pp. 505–536.

Dilkina, Bistra, Carla P Gomes, Yuri Malitsky, et al. (2009). “Backdoors to combinatorial optimization: Feasibility and optimality”. In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer, pp. 56–70.

Dilkina, Bistra, Carla P Gomes, and Ashish Sabharwal (2009). “Backdoors in the context of learning”. In: *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*.

Dorigo, Marco, Mauro Birattari, and Thomas Stutzle (2006). “Ant colony optimization”. In: *IEEE computational intelligence magazine* 1.4, pp. 28–39.

Douguet, Dominique, Etienne Thoreau, and Gérard Grassy (2000). “A genetic algorithm for the automated generation of small organic molecules: drug design using an evolutionary algorithm”. In: *Journal of computer-aided molecular design* 14.5, pp. 449–466.

Du, Dingzhu and Panos M Pardalos (1998). *Handbook of combinatorial optimization*. Vol. 4. Springer Science & Business Media.

Duan, Yan et al. (2016). “Benchmarking deep reinforcement learning for continuous control”. In: *International Conference on Machine Learning*.

Dvořák, Pavel et al. (2017). “Solving integer linear programs with a small number of global variables and constraints”. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pp. 607–613.

Erdős, Paul and Alfréd Rényi (1960). “On the evolution of random graphs”. In: *Publ. Math. Inst. Hung. Acad. Sci*.

Ferber, Aaron, Jialin Song, Bistra Dilkina, and Yisong Yue (2021). “Learning Pseudo-Backdoors for Mixed Integer Programs”. In: *Symposium on Combinatorial Search*.

Ferber, Aaron, Bryan Wilder, et al. (2020). “Mipaal: Mixed integer program as a layer”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 02, pp. 1504–1511.

Fischetti, Matteo and Andrea Lodi (2010). “Heuristics in mixed integer programming”. In: *Wiley Encyclopedia of Operations Research and Management Science*.

Fischetti, Matteo and Michele Monaci (2011). “Backdoor branching”. In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer, pp. 183–191.

Gal, Yarin and Zoubin Ghahramani (June 2016). “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, pp. 1050–1059. URL: `http://proceedings.mlr.press/v48/gal16.html`.

Galef Jr, Bennett G (1988). “Imitation in animals: History, definition, and interpretation of data from the psychological laboratory”. In: *Social learning: Psychological and biological perspectives* 28.

Gasse, Maxime et al. (2019). “Exact Combinatorial Optimization with Graph Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 32*.

Golombek, M and Don Rapp (1997). “Size-frequency distributions of rocks on Mars and Earth analog sites: Implications for future landed missions”. In: *Journal of Geophysical Research: Planets* 102.E2, pp. 4117–4129.

Golovin, Daniel and Andreas Krause (2011). “Adaptive submodularity: Theory and applications in active learning and stochastic optimization”. In: *Journal of Artificial Intelligence Research* 42, pp. 427–486.

Gomes, Carla P and Bart Selman (2001). “Algorithm portfolios”. In: *Artificial Intelligence* 126.1-2, pp. 43–62.

Gonen, Rica and Daniel Lehmann (2000). “Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics”. In: *ACM Conference on Economics and Computation (EC)*.

Greensmith, Evan, Peter L Bartlett, and Jonathan Baxter (2004). “Variance reduction techniques for gradient estimates in reinforcement learning”. In: *Journal of Machine Learning Research*.

Gu, Shixiang et al. (2017). “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates”. In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, pp. 3389–3396.

Guillaume, Alexandre et al. (2007). “Deep space network scheduling using evolutionary computational methods”. In: *2007 IEEE Aerospace Conference*. IEEE, pp. 1–6.

Gurobi Optimization, LLC (2021). *Gurobi Optimizer Reference Manual*. URL: `http://www.gurobi.com`.

Haarnoja, Tuomas et al. (2019). “Learning to Walk Via Deep Reinforcement Learning.” In: *Robotics: Science and Systems*.

Hart, Peter E, Nils J Nilsson, and Bertram Raphael (1968). “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

He, He, Hal Daume III, and Jason M Eisner (2014). “Learning to Search in Branch and Bound Algorithms”. In: *Neural Information Processing Systems (NeurIPS)*.

Helber, Stefan and Florian Sahling (2010). “A fix-and-optimize approach for the multi-level capacitated lot sizing problem”. In: *International Journal of Production Economics* 123.2, pp. 247–256.

Hendel, Gregor (2018). “Adaptive large neighborhood search for mixed integer programming”. In:

Henderson, Peter et al. (2018). “Deep reinforcement learning that matters”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

Hester, Todd et al. (2018). “Deep Q-Learning from Demonstrations”. In: *AAAI Conference on Artificial Intelligence*.

Heyes, Cecilia M and Bennett G Galef Jr (1996). *Social learning in animals: the roots of culture*. Elsevier.

Ho, Jonathan and Stefano Ermon (2016). “Generative adversarial imitation learning”. In: *Neural Information Processing Systems*.

Hochbaum, Dorit S and Anu Pathria (1997). “Forest harvesting and minimum cuts: a new approach to handling spatial constraints”. In: *Forest Science* 43.4, pp. 544–554.

Holmberg, Kaj and Di Yuan (2000). “A Lagrangian heuristic based branch-and-bound approach for the capacitated network design problem”. In: *Operations Research* 48.3, pp. 461–481.

Hoos, Holger H (2011). “Automated algorithm configuration and parameter tuning”. In: *Autonomous search*. Springer, pp. 37–71.

Hottung, André and Kevin Tierney (2019). “Neural Large Neighborhood Search for the Capacitated Vehicle Routing Problem”. In: *arXiv:1911.09539*.

Huang, Taoan, Bistra Dilkina, and Sven Koenig (2021). “Learning Node-Selection Strategies in Bounded-Suboptimal Conflict-Based Search for Multi-Agent Path Finding”. In: *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 611–619.

Hutter, Frank, Holger H Hoos, and Kevin Leyton-Brown (2011). “Sequential model-based optimization for general algorithm configuration”. In: *International conference on learning and intelligent optimization*. Springer, pp. 507–523.

Ichter, Brian, James Harrison, and Marco Pavone (2018). “Learning sampling distributions for robot motion planning”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 7087–7094.

Jauhri, Snehal, Carlos Celemin, and Jens Kober (2020). “Interactive Imitation Learning in State-Space”. In: *arXiv preprint arXiv:2008.00524*.

Javdani, Shervin et al. (Apr. 2014). “Near-Optimal Bayesian Active Learning for Decision Making”. In: *In Proc. International Conference on Artificial Intelligence and Statistics (AISTATS)*.

Johnson, Melvin et al. (2017). “Google’s multilingual neural machine translation system: Enabling zero-shot translation”. In: *Transactions of the Association for Computational Linguistics*.

Joshi, Chaitanya K, Thomas Laurent, and Xavier Bresson (2019). "An efficient graph convolutional network technique for the travelling salesman problem". In: *arXiv preprint arXiv:1906.01227*.

Kakade, Sham and John Langford (2002). "Approximately Optimal Approximate Reinforcement Learning". In: *International Conference on Machine Learning*.

Kang, Bingyi, Zequn Jie, and Jiashi Feng (2018). "Policy optimization with demonstrations". In: *International Conference on Machine Learning*.

Khalil, Elias, Hanjun Dai, et al. (2017). "Learning combinatorial optimization algorithms over graphs". In: *NeurIPS*.

Khalil, Elias, Pierre Le Bodic, et al. (2016). "Learning to branch in mixed integer programming". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1.

Khalil, Elias B et al. (2017). "Learning to Run Heuristics in Tree Search". In: *IJCAI*.

Kilby, Philip et al. (2005). "Backbones and backdoors in satisfiability". In: *Proceedings of the 20th national conference on Artificial intelligence-Volume 3*, pp. 1368–1373.

Kim, Yoon et al. (2018). "Semi-amortized variational autoencoders". In: *ICML*.

Kleinberg, Robert et al. (2019). "Procrastinating with Confidence: Near-Optimal, Anytime, Adaptive Algorithm Configuration". In: *NeurIPS*.

Kober, Jens, J Andrew Bagnell, and Jan Peters (2013). "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research*.

Koch, Thorsten et al. (2011). "MIPLIB 2010". In: *Mathematical Programming Computation* 3.2, pp. 103–163.

Kool, Wouter, Herke van Hoof, and Max Welling (2018). "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations*.

Kruskal, Joseph B (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". In: *Proceedings of the American Mathematical society* 7.1, pp. 48–50.

Kumar, Abhishek and Hal Daumé (2011). "A co-training approach for multi-view spectral clustering". In: *International Conference on Machine Learning*.

Land, Ailsa H and Alison G Doig (2010). "An automatic method for solving discrete programming problems". In: *50 Years of Integer Programming 1958-2008*. Springer, pp. 105–132.

Lawler, Eugene L and David E Wood (1966). "Branch-and-bound methods: A survey". In: *Operations research* 14.4, pp. 699–719.

Le, Hoang et al. (2018). "Hierarchical Imitation and Reinforcement Learning". In: *International Conference on Machine Learning*.

Le Bodic, Pierre and George L Nemhauser (2015). “How important are branching decisions: fooling MIP solvers”. In: *Operations Research Letters* 43.3, pp. 273–278.

Levine, Sergey et al. (2016). “End-to-end training of deep visuomotor policies”. In: *The Journal of Machine Learning Research*.

Leyton-Brown, Kevin, Mark Pearson, and Yoav Shoham (2000). “Towards a universal test suite for combinatorial auction algorithms”. In: *ACM conference on Electronic commerce*, pp. 66–76.

Li, Yujia et al. (Apr. 2016). “Gated Graph Sequence Neural Networks”. In: *Proceedings of ICLR’16*.

Lillicrap, Timothy P et al. (2015). “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971*.

Lin, Hui and Jeff Bilmes (2012). “Learning mixtures of submodular shells with application to document summarization”. In: *Conference on Uncertainty in Artificial Intelligence (UAI)*.

Linderoth, Jeff (2005). “A simplicial branch-and-bound algorithm for solving quadratically constrained quadratic programs”. In: *Mathematical programming*.

Linderoth, Jeff T and Martin WP Savelsbergh (1999). “A computational study of search strategies for mixed integer programming”. In: *INFORMS Journal on Computing* 11.2, pp. 173–187.

Lindley, Dennis V (1956). “On a measure of the information provided by an experiment”. In: *The Annals of Mathematical Statistics*, pp. 986–1005.

Liu, Ming, Wray Buntine, and Gholamreza Haffari (2018). “Learning how to actively learn: A deep imitation learning approach”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1874–1883.

Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng (2013). “Rectifier nonlinearities improve neural network acoustic models”. In: *International Conference on Machine Learning (ICML)*.

Manna, Zohar and Richard J Waldinger (1971). “Toward automatic program synthesis”. In: *Communications of the ACM* 14.3, pp. 151–165.

Mirhoseini, Azalia et al. (2017). “Device placement optimization with reinforcement learning”. In: *International Conference on Machine Learning (ICML)*.

Mittelmann, Hans D (2017). “Latest Benchmarks of Optimization Software”. In:

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, et al. (2013). “Playing atari with deep reinforcement learning”. In: *arXiv*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, et al. (2015). “Human-level control through deep reinforcement learning”. In: *nature* 518.7540, pp. 529–533.

Nair, Ashvin et al. (2018). “Overcoming exploration in reinforcement learning with demonstrations”. In: *International Conference on Robotics and Automation*.

Nehaniv, Chrystopher L and Kerstin Ed Dautenhahn (2007). *Imitation and social learning in robots, humans and animals: behavioural, social and communicative dimensions.* Cambridge University Press.

Ng, Andrew Y, Daishi Harada, and Stuart J Russell (1999). “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping”. In: *Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 278–287.

Nigam, Kamal and Rayid Ghani (2000). “Analyzing the effectiveness and applicability of co-training”. In: *ACM Conference on Information and knowledge Management*.

Öncan, Temel, İ Kuban Altınel, and Gilbert Laporte (2009). “A comparative analysis of several asymmetric traveling salesman problem formulations”. In: *Computers & Operations Research*.

Ono, Masahiro and Brian C Williams (2008). “An Efficient Motion Planning Algorithm for Stochastic Dynamic Systems with Constraints on Probability of Failure.” In: *AAAI*, pp. 1376–1382.

Ono, Masahiro, Brian C Williams, and Lars Blackmore (2013). “Probabilistic planning for continuous dynamic systems under bounded risk”. In: *Journal of Artificial Intelligence Research (JAIR)* 46, pp. 511–577.

Optimization, Gurobi (2019). *Gurobi 8 Performance Benchmarks*. URL: https://www.gurobi.com/pdfs/benchmarks.pdf.

Orman, AJ and HP Williams (2007). “A survey of different integer programming formulations of the travelling salesman problem”. In: *Optimisation, econometric and financial analysis*. Springer.

Otsu, Kyohei et al. (2020). “Fast approximate clearance evaluation for rovers with articulated suspension systems”. In: *Journal of Field Robotics* 37.5, pp. 768–785. DOI: 10.1002/rob.21892. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21892. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%20rob.21892.

Paliwal, Aditya et al. (2020). “Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=rkxDoJBYPB.

Paszke, Adam et al. (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Peng, Fei et al. (2010). "Population-based algorithm portfolios for numerical optimization". In: *IEEE Transactions on evolutionary computation* 14.5, pp. 782–800.

Pisinger, David and Stefan Ropke (2010). "Large neighborhood search". In: *Handbook of metaheuristics*.

Pomerleau, Dean A (1989). "Alvinn: An autonomous land vehicle in a neural network". In: *NeurIPS*.

Prékopa, András (1999). "The Use of Discrete Moment Bounds in Probabilistic Constrained Stochastic Programming Models". In: *Annals of Operations Research* 85, pp. 21–38.

Prouvost, Antoine et al. (2020). "Ecole: A Gym-like Library for Machine Learning in Combinatorial Optimization Solvers". In: *Learning Meets Combinatorial Algorithms at NeurIPS2020*. URL: `https://openreview.net/forum?id=IVc9hqgibyB`.

Puterman, Martin L (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Qin, Tao, Tie-yan Liu Jun, and Xu Hang (2010). "LETOR : A Benchmark Collection for Research on Learning to Rank for Information Retrieval". In: *Information Retrieval* 13(4), pp. 346–374.

Quigley, Morgan et al. (2009). "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan, p. 5.

Rao, Roshan et al. (Dec. 2019). "Evaluating Protein Transfer Learning with TAPE". eng. In: *Advances in neural information processing systems* 32. PMC7774645[pmcid], pp. 9689–9701. ISSN: 1049-5258. URL: `https://pubmed.ncbi.nlm.nih.gov/33390682`.

Rolínek, Michal et al. (2020). "Deep graph matching via blackbox differentiation of combinatorial solvers". In: *European Conference on Computer Vision*. Springer, pp. 407–424.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (2015). "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer, pp. 234–241.

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

Ross, Stephane, Jiaji Zhou, et al. (2013). "Learning Policies for Contextual Submodular Prediction". In: *International Conference on Machine Learning*, pp. 1364–1372.

Ross, Stéphane and Drew Bagnell (2010). "Efficient reductions for imitation learning". In: pp. 661–668.

Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell (2011). “A reduction of imitation learning and structured prediction to no-regret online learning”. In: *International Conference on Artificial Intelligence and Statistics*.

Rothberg, Edward (2007). “An evolutionary algorithm for polishing mixed integer programming solutions”. In: *INFORMS Journal on Computing* 19.4, pp. 534–541.

Runge, M. C., S. J. Converse, and J. E. Lyons (2011). “Which uncertainty? Using expert elicitation and expected value of information to design an adaptive program”. In: *Biological Conservation*.

Satsangi, Yash et al. (2018). “Exploiting submodular value functions for scaling up active perception”. In: *Autonomous Robots* 42.2, pp. 209–233.

Schouwenaars, Tom et al. (2001). “Mixed Integer Programming for Multi-Vehicle Path Planning”. In: *European Control Conference*, pp. 2603–2608.

Schulman, John, Sergey Levine, et al. (2015). “Trust region policy optimization”. In: *International Conference on Machine Learning*.

Schulman, John, Filip Wolski, et al. (2017). “Proximal policy optimization algorithms”. In: *arXiv*.

Shaw, Paul (1998). “Using constraint programming and local search methods to solve vehicle routing problems”. In: *CP*.

Silver, David et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587, pp. 484–489.

Singla, Adish et al. (2014). “Near-Optimally Teaching the Crowd to Classify.” In: *International Conference on Machine Learning (ICML)*.

Sjöbring, U., L. Björck, and W. Kastern (Jan. 1991). “Streptococcal protein G. Gene structure and protein binding properties”. In: *J. Biol. Chem.* 266.1, pp. 399–405.

Song, Jialin, Yuxin Chen, and Yisong Yue (2019). “A general framework for multi-fidelity bayesian optimization with gaussian processes”. In: *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 3158–3167.

Song, Jialin, Ravi Lanka, Yisong Yue, and Bistra Dilkina (2020). “A General Large Neighborhood Search Framework for Solving Integer Linear Programs”. In: *Advances in Neural Information Processing Systems*. Vol. 33. URL: https://proceedings.neurips.cc/paper/2020/hash/e769e03a9d329b2e864b4bf4ff54ff39-Abstract.html.

Song, Jialin, Ravi Lanka, Yisong Yue, and Masahiro Ono (2020). “Co-training for policy learning”. In: *Uncertainty in Artificial Intelligence*. PMLR, pp. 1191–1201. URL: http://proceedings.mlr.press/v115/song20b.html.

Song, Jialin, Ravi Lanka, Albert Zhao, Aadyot Bhatnagar, Yisong Yue, and Masahiro Ono (2018). “Learning to search via retrospective imitation”. In: *arXiv preprint*. URL: `https://arxiv.org/abs/1804.00846`.

Sousa, Samuel de, Yll Haxhimusa, and Walter G Kropatsch (2013). “Estimation of distribution algorithm for the max-cut problem”. In: *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer, pp. 244–253.

Stadie, Bradly C, Pieter Abbeel, and Ilya Sutskever (2017). “Third-person imitation learning”. In: *arXiv*.

Sun, Bo et al. (2020). “Competitive Algorithms for the Online Multiple Knapsack Problem with Application to Electric Vehicle Charging”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3, pp. 1–32.

Sun, Wen et al. (2017). “Deeply aggrevated: Differentiable imitation learning for sequential prediction”. In: *International Conference on Machine Learning*.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, Richard S, David A McAllester, et al. (2000). “Policy gradient methods for reinforcement learning with function approximation”. In: *Neural information processing systems*.

Syed, Arslan Ali et al. (2019). “Neural network based large neighborhood search algorithm for ride hailing services”. In: *EPIA*.

Syed, Umar, Michael Bowling, and Robert E Schapire (2008). “Apprenticeship learning using linear programming”. In: *International Conference on Machine Learning*.

Syed, Umar and Robert E Schapire (2008). “A game-theoretic approach to apprenticeship learning”. In: *Neural Information Processing Systems (NeurIPS)*.

Tang, Yunhao, Shipra Agrawal, and Yuri Faenza (2020). “Reinforcement learning for integer programming: Learning to cut”. In: *International Conference on Machine Learning*. PMLR, pp. 9367–9376.

Todorov, Emanuel, Tom Erez, and Yuval Tassa (2012). “Mujoco: A physics engine for model-based control”. In: *International Conference on Intelligent Robots and Systems*.

Toupet, Olivier et al. (2020). “A ROS-based Simulator for Testing the Enhanced Autonomous Navigation of the Mars 2020 Rover”. In: *IEEE Aerospace*.

Tsochantaridis, Ioannis et al. (2005). “Large margin methods for structured and interdependent output variables.” In: *Journal of machine learning research* 6.9.

Vega, Wenceslas Fernandez de la and Claire Kenyon-Mathieu (2007). “Linear programming relaxations of maxcut”. In: *ACM-SIAM symposium on Discrete algorithms*.

Veličković, Petar et al. (2018). “Graph Attention Networks”. In: *International Conference on Learning Representations*. accepted as poster. URL: `https://openreview.net/forum?id=rJXMpikCZ`.

Vlastelica, Marin et al. (2019). “Differentiation of blackbox combinatorial solvers”. In: *International Conference on Learning Representations*.

Wainwright, Martin, Tommi Jaakkola, and Alan Willsky (2005). “MAP estimation via agreement on trees: message-passing and linear programming”. In: *IEEE transactions on information theory*.

Wan, Xiaojun (2009). “Co-training for cross-lingual sentiment classification”. In: *Joint conference of ACL and IJCNLP*. Association for Computational Linguistics.

Wang, C. Y. et al. (Mar. 2019). “ProtaBank: A repository for protein design and engineering data”. In: *Protein Sci.* 28.3, p. 672.

Wang, Po-Wei et al. (2019). “Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver”. In: *International Conference on Machine Learning*. PMLR, pp. 6545–6554.

Wilder, Bryan et al. (2019). “End to End Learning and Optimization on Graphs”. In: *Advances in Neural and Information Processing Systems*.

Williams, Ryan, Carla P Gomes, and Bart Selman (2003). “Backdoors to typical case complexity”. In: *Proceedings of the 18th international joint conference on Artificial intelligence*, pp. 1173–1178.

Williford, Kenneth H et al. (2018). “The NASA Mars 2020 rover mission and the search for extraterrestrial life”. In: *From Habitability to Life on Mars*. Elsevier, pp. 275–308.

Xu, Lin et al. (2008). “SATzilla: portfolio-based algorithm selection for SAT”. In: *Journal of artificial intelligence research* 32, pp. 565–606.

Zhang, Lintao and Sharad Malik (2002). “The quest for efficient boolean satisfiability solvers”. In: *CAV*.

Ziebart, Brian et al. (2008). “Maximum entropy inverse reinforcement learning”. In: *AAAI Conference on Artificial Intelligence*.

Appendix A

APPENDIX TO CHAPTER 4

A.1 Proofs

Proof for Proposition 6:

Proof. We show that $\max_s D_{JS}(\pi^B(s)\|\pi^A(s))$ is well-defined for an MDP $\mathcal{M}$ with two representations $\mathcal{M}^A$ and $\mathcal{M}^B$. From Theorem 4, we know the distribution $\pi(s)$ can be written with respect to its occupancy measure ρ_π. It is sufficient to show that we can map occupancy measures of π^A and π^B to a common MDP. By the definition of an occupancy measure,

$$\begin{aligned}\rho_\pi(s,a) &= \mathbb{P}(\pi(s)=a)\sum\nolimits_{i=0}^{\infty}\gamma^i\mathbb{P}(s_i=s|\pi)\\ &= \mathbb{E}_{\tau=(s_0,a_0,\cdots,s_n)\sim\pi}[\sum\nolimits_{i=0}^{n}\gamma^i\mathbf{1}((s_i,a_i)=(s,a))].\end{aligned}$$

that is to say, the occupancy measure is the expected discounted count of a state-action pair to appear in all possible trajectories. Since we have trajectory mappings between $\mathcal{M}^A$ and $\mathcal{M}^B$, we can convert an occupancy measure in $\mathcal{M}^A$ to one in $\mathcal{M}^B$ by mapping each trajectory and perform the count in the new MDP representation. Formally, the occupancy measure $\rho^B_{\pi^B}$ of π^B in $\mathcal{M}^B$ can be mapped to an occupancy measure in $\mathcal{M}^A$ by

$$\begin{aligned}&\rho^A_{\pi^B}(s,a)\\ &= \mathbb{E}_{\substack{\tau^B\sim\pi^B,\\ f_{B\to A}(\tau^B)=(s_0,a_0,\cdots,s_n)}}[\sum\nolimits_{i=0}^{n}\gamma^i\mathbf{1}((s_i,a_i)=(s,a))].\end{aligned}$$

Following from this, we can compute $D_{JS}(\pi^B(s)\|\pi^A(s))$ using any s in $\mathcal{M}^A$. And the maximum is defined. In the definition, there is a choice whether to map π^A's occupancy measure to $\mathcal{M}^B$ or π^B's to $\mathcal{M}^A$. Though both approaches lead to a valid definition, we use the definition that for $D_{JS}(\cdot\|\cdot)$, we always map the representation in the first argument to that of the second argument. It is preferable to the other one because in Theorem 5, we want to optimize

$$J(\pi'^A) \geq J_{\pi^A}(\pi'^A) - \frac{2\gamma^A(4\beta^B_{\mathcal{D}_2}\epsilon^B_{\mathcal{D}_2}+\alpha^A_{\mathcal{D}}\epsilon^A_{\mathcal{D}})}{(1-\gamma^A)^2} + \delta_2$$

by optimizing

$$\beta^B_{\mathcal{D}_2} = \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\max_s D_{JS}(\pi^B(s)\|\pi^A(s))].$$

usually via computing the gradient of $\beta^B_{\mathcal{D}_2}$ w.r.t. π^A. If we use $f_{A\to B}$ to map from $\mathcal{M}^A$ to $\mathcal{M}^B$, the gradient will involve a complex composition of $f_{A\to B}$ and π^A, which is undesirable. $\square$

To prove Theorem 5, we need to use a policy improvement result for a single MDP (a modified version of Theorem 1 in (Kang, Jie, and Feng, 2018)).

Theorem 10. *Assume for an MDP $\mathcal{M}$, an expert policy π_E have a higer advantage of over a policy π with a margin, i.e., $\eta(\pi_E, \mathcal{M}) - \eta(\pi, \mathcal{M}) \geq \delta$ Define*

$$\begin{aligned} \alpha &= \max\nolimits_s D_{KL}(\pi'(s) \| \pi(s)) \\ \beta &= \max\nolimits_s D_{JS}(\pi'(s) \| \pi_E(s)) \\ \epsilon_{\pi_E} &= \max\nolimits_{s,a} |A_{\pi_E}(s,a)| \\ \epsilon_{\pi} &= \max\nolimits_{s,a} |A_{\pi}(s,a)| \end{aligned}$$

then $\eta(\pi', \mathcal{M}) \geq \eta_\pi(\pi', \mathcal{M}) - \frac{2\gamma(4\beta\epsilon_{\pi_E} + \alpha\epsilon_\pi)}{(1-\gamma)^2} + \delta$.

Proof. The only difference from the original theorem is that the original assumes $\mathbb{E}_{a_E \sim \pi_E(s), a\sim\pi(s)}[A_\pi(s, a_E) - A_\pi(s,a)] \geq \delta' > 0$ for every state s. It is a stronger assumption which is not needed in their analysis. Notice that the advantage of a policy over itself is zero, i.e., $\mathbb{E}_{a\sim\pi(s)}[A_\pi(s,a)] = 0$ for every s, so the margin assumption simplifies to $\mathbb{E}_{a_E\sim\pi_E(s)}[A_\pi(s,a_E)] \geq \delta'$.

By the policy advantage formula,

$$\begin{aligned} \eta(\pi_E, \mathcal{M}) - \eta(\pi, \mathcal{M}) &= \mathbb{E}_{\tau\sim\pi_E}[\sum\nolimits_{i=0}^{\infty} \gamma^i A_\pi(s_i, a_i)] \\ &= \mathbb{E}_{s_i\sim\rho_{\pi_E}} \mathbb{E}_{a_i\sim\pi_E(s_i)}[\sum\nolimits_{i=0}^{\infty} \gamma^i A_\pi(s_i, a_i)] \\ &\geq \mathbb{E}_{s_i\sim\rho_{\pi_E}}[\delta' \sum\nolimits_{i=0}^{\infty} \gamma^i] \\ &= \frac{\delta'}{1-\gamma}. \end{aligned}$$

So an assumption on per-state advantage translates to a overall advantage. Thus we can make this weaker assumption which is also more intuitive and the original statement still holds with a different δ term. $\square$

Proof of Theorem 5:

Proof. Theorem 5 is a distributional extension to the theorem above. For $\mathcal{M} \sim \mathcal{D}_2$, let $\delta_\mathcal{M} = \eta(\pi^B, \mathcal{M}^B) - \eta(\pi^A, \mathcal{M}^A)$.

$$\begin{aligned}
&J(\pi'^A) \\
&= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}}[\eta(\pi'^A, \mathcal{M}^A)] \\
&= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_1}[\eta(\pi'^A, \mathcal{M}^A)] + \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\eta(\pi'^A, \mathcal{M}^A)] \\
&\geq \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_1}[\eta(\pi'^A, \mathcal{M}^A)] + \\
&\mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\eta_{\pi^A}(\pi'^A, \mathcal{M}^A) - \frac{2\gamma^A(4\beta\epsilon_{\pi^B} + \alpha\epsilon_{\pi^A})}{(1-\gamma^A)^2} + \delta_\mathcal{M}] \\
&\geq \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_1}[\eta_{\pi^A(}(\pi'^A, \mathcal{M}^A) - \frac{2\gamma^A\alpha\epsilon_{\pi^A}}{(1-\gamma^A)^2}] + \\
&\mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\eta_{\pi^A}(\pi'^A, \mathcal{M}^A) - \frac{2\gamma^A(4\beta\epsilon_{\pi^B} + \alpha\epsilon_{\pi^A})}{(1-\gamma^A)^2} + \delta_\mathcal{M}] \\
&= \mathbb{E}_{\mathcal{M}\sim\mathcal{D}}[\eta_{\pi^A}(\pi'^A, \mathcal{M}^A)] - \mathbb{E}_{\mathcal{M}\sim\mathcal{D}}[\frac{2\gamma^A\alpha\epsilon_{\pi^A}}{(1-\gamma^A)^2}] - \\
&\mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\frac{2\gamma^A \cdot 4\beta\epsilon_{\pi^B}}{(1-\gamma^A)^2}] + \mathbb{E}_{\mathcal{M}\sim\mathcal{D}_2}[\delta_\mathcal{M}] \\
&\geq J_{\pi^A}(\pi'^A) - \frac{2\gamma^A(4\beta^B_{\mathcal{D}_2}\epsilon^B_{\mathcal{D}_2} + \alpha^A_\mathcal{D}\epsilon^A_\mathcal{D})}{(1-\gamma^A)^2} + \delta_2.
\end{aligned}$$

The derivation for $J(\pi'^B)$ is the same. □

Finally, we provide the proof for Theorem 7. We first quantify the performance gap between a policy π and an optimal policy π^*. For a policy that is able to achieve ϵ $0-1$ loss, $\ell(s,\pi) = \mathbf{1}(\pi(s) \neq \pi^*(s))$, measured against π^*'s action choices under its own state distributions, then we can bound the performance gap. Let $Q^{\pi'}_t(s,\pi)$ denote the t-step cost of executing π in initial state s and then following policy π'

Theorem 11. *(Theorem 2.2 from (Stéphane Ross, Gordon, and D. Bagnell, 2011), adpated to our notations) Let* π *be such taht* $\mathbb{E}_{s\sim\rho_\pi}[\ell(s,\pi)] = \epsilon$*, and* $Q^{\pi^*}_{T-t+1}(s,\pi^*) - Q^{\pi^*}_{T-t+1}(s,a) \leq u$ *for all action* $a, t \in \{1, 2, \cdots, T\}$*, then* $\eta(\pi, \mathcal{M}) \geq \eta(\pi^*, \mathcal{M}) - uT\epsilon$*.*

Thus the important quantity to measure is ϵ, and by measuring the disagreements between two policies in two views, we can upper bound ϵ. The result is originally stated in the context of classification, and the above theorem justifies the learning reduction approach of reducing policy learning to classification.

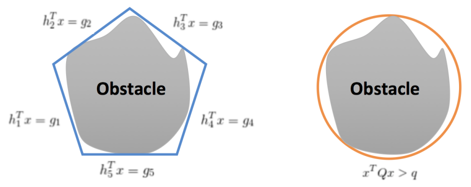


Figure A.1: Two views for Risk-Aware Path Planning. On the left, the obstacle is enclosed by a polytope (MILP view) and on the right the obstacle is enclosed by an ellipse (QCQP view).

Theorem 12. *(Corollary 5 in (Dasgupta, Littman, and McAllester, 2002) applied to full classifiers) Using the definitions in Theorem 7, with probability* $1 - \sigma$ *over the choice of a sample set* N*, for all pairs of classifiers* h_1, h_2 *such that for all* i *we have* $\zeta_i(h_1, h_2, \sigma) > 0$ *and* $b_i(h_1, h_2, \sigma) \leq 1$.

$$\epsilon \leq \max_{j \in \{1, \cdots, k\}} b_j(h_1, h_2, \sigma)$$

Proof. The only change from the original proof is that instead of a partial classifier which can output $\perp$, we consider a full classifier. Then we could eliminate the estimates for $\mathbb{P}(h_1 \neq \perp)$ and the error introduced by converting a partial classifier to a full classifier via random labelling when the output is $\perp$. □

Proof of Theorem 7:

Proof. For the bound for π^A, we are measuring ϵ_A on its sampled paths. Then directly apply Theorem 12 gives an upper bound on ϵ_A. Apply Theorem 11 gives the result of Theorem 7. □

A.2 Pictorial Representation of the Two-views in Risk-aware Path Planning

We present a pictorial representation of the two different views used in the experiments in Fig A.1. In the MILP view, the constraint space is represented using additional auxiliary binary variables to choose the active side of the polytope, whereas in the QCQP view, the constraint space can be encoded in a quadratic constraint.

A.3 Risk-aware Path Planning Dataset Generation

We generate 150 obstacle maps. Each map contains 10 rectangle obstacles, with the center of each obstacle chosen from a uniform random distribution over the space $0 \leq y \leq 1, 0 \leq x \leq 1$. The side length of each obstacle was chosen from a uniform distribution in range [0.01, 0.02] and the orientation was chosen from a uniform distribution between $0°$ and $360°$. In order to avoid trivial infeasible maps, any obstacles centered close to the destination are removed. For MILP view, we directly use the randomly generated rectangles for defining the constraint space. However, for the QCQP view, we enclose the rectangle obstacles with a circle for defining the quadratic constraint.

A.4 Discrete/Continuous Control Results in Tabular Form

	Acrobot	Swimmer	Hopper
A (CoPiEr)	-86.44 ± 10.80	106.35 ± 23.11	217.83 ± 30.03
A (PG)	-169.57 ± 10.48	109.09 ± 21.58	278.66 ± 32.87
A (All)	-252.42 ± 8.73	100.36 ± 22.37	49.39 ± 10.35
B (CoPiEr)	-88.48 ± 15.13	104.16 ± 19.32	168.88 ± 18.21
B (PG)	-257.16 ± 10.93	103.48 ± 21.89	89.34 ± 4.89
B (All)	-251.74 ± 9.65	96.74 ± 19.57	22.59 ± 5.55
A + B	-86.42 ± 3.48	108.71 ± 5.03	346.53 ± 5.91

Appendix B

APPENDIX TO CHAPTER 5

B.1 Algorithm Configuration Results

In this section, we present the algorithm configuration results similar to the one in Section 5.4.

k \ t	1	2	3
2	13.61 ± 0.82	14.19 ± 0.89	$\mathbf{14.42 \pm 0.88}$
3	6.06 ± 0.47	6.17 ± 0.42	6.65 ± 0.45
4	3.09 ± 0.30	3.14 ± 0.27	3.61 ± 0.31
5	1.84 ± 0.18	2.13 ± 0.20	2.08 ± 0.23

Table B.1: Parameter sweep results for (k, t) of an MVC dataset for Erdős-Rényi random graphs with 1000 vertices. Numbers represent improvement ratios Δ/t for one decomposition, averaged over 5 random seeds.

k \ t	1	2	3
2	69.05 ± 2.92	71.87 ± 2.98	$\mathbf{74.14 \pm 3.03}$
3	29.99 ± 2.07	28.59 ± 1.80	30.05 ± 1.81
4	14.28 ± 1.04	16.13 ± 1.13	15.33 ± 1.19
5	7.79 ± 0.77	7.57 ± 0.72	7.69 ± 0.69

Table B.2: Parameter sweep results for (k, t) of the MVC dataset for Barabási-Albert random graphs with 1000 vertices.

k \ t	1	2	3
2	2155.60 ± 22.79	1258.79 ± 13.65	925.05 ± 12.50
3	2700.33 ± 20.91	1767.37 ± 10.86	1310.96 ± 6.25
4	4454.65 ± 46.05	4489.60 ± 49.44	4466.36 ± 47.20
5	$\mathbf{5414.01 \pm 29, 76}$	5325.95 ± 31.07	5404.87 ± 30.16

Table B.3: Parameter sweep results for (k, t) of the MAXCUT dataset for Erdős-Rényi random graphs with 500 vertices.

B.2 Visualization

A natural question is what property a good decomposition has. Here we provide one interpretation for the risk-aware path planning. We use a slightly smaller instance

t k	1	2	3
2	1961.42 ± 24.54	1043.89 ± 12.28	1030.60 ± 2.41
3	2698.46 ± 36.44	1887.29 ± 51.40	1581.43 ± 55.98
4	6565.54 ± 47.36	6454.62 ± 46.80	$\mathbf{6669.28 \pm 47.91}$
5	6400.38 ± 23.94	6478.23 ± 19.33	6465.03 ± 22.54

Table B.4: Parameter sweep results for (k, t) of the MAXCUT dataset for Barabási-Albert random graphs with 500 vertices.

t k	1	2	3
2	$\mathbf{65360.28 \pm 799.26}$	37554.81 ± 263.48	27864.92 ± 179.93
3	61064.41 ± 519.66	36816.46 ± 236.11	26633.11 ± 178.71
4	56190.18 ± 530.23	34647.30 ± 233.18	25547.98 ± 176.94
5	54571.21 ± 344.89	33554.38 ± 224.77	24238.73 ± 165.66

Table B.5: Parameter sweep results for (k, t) of the CATS dataset for the regions distribution with 2000 items and 4000 bids.

t k	1	2	3
2	$\mathbf{54358.95 \pm 1268.30}$	31397.88 ± 364.19	21878.70 ± 234.63
3	50046.53 ± 586.72	29375.81 ± 336.84	20711.09 ± 242.39
4	46449.07 ± 555.02	27920.03 ± 315.03	20431.02 ± 226.95
5	42190.19 ± 480.57	27004.79 ± 315.24	19882.16 ± 211.44

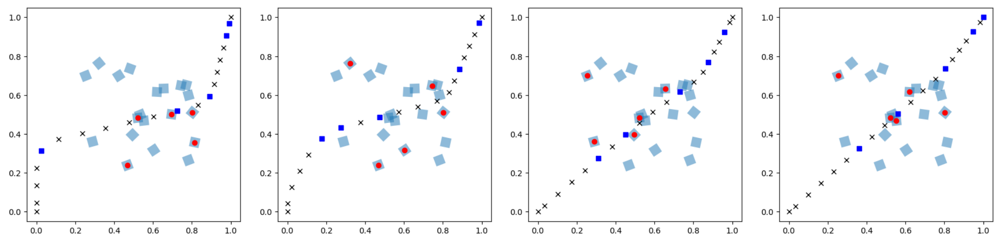
Table B.6: Parameter sweep results for (k, t) of the CATS dataset for the arbitrary distribution with 2000 items and 4000 bids.

t k	1	2	3
2	0.37 ± 0.18	0.39 ± 0.07	0.36 ± 0.05
3	0.41 ± 0.07	$\mathbf{0.43 \pm 0.07}$	0.43 ± 0.07
4	0.37 ± 0.06	0.40 ± 0.06	0.33 ± 0.05
5	0.33 ± 0.04	0.32 ± 0.05	0.31 ± 0.05

Table B.7: Parameter sweep results for (k, t) of the risk-aware path planning for 30 obstacles.

with 20 obstacles for a clearer view. Binary variables in an ILP formulation of this problem model relationships between obstacles and waypoints. Thus we can interpret the neighborhood formed by a subset of binary variables as attention over specific relationships among some obstacles and waypoints.

Figure B.1 captures 4 consecutive iterations of LNS with large solution improve-



(a) Iteration 2 (b) Iteration 3 (c) Iteration 4 (d) Iteration 5

Figure B.1: Visualizing predicted decompositions in a risk-aware path planning problem, with 4 consecutive solutions after 3 iterations of LNS. Each blue square is an obstacle and each cross is a waypoint. The obstacles in red and waypoints in dark blue are the most frequent ones in the subsets that lead to high local improvement.

ments. Each sub-figure contains information about the locations of obstacles (light blue squares) and the waypoint locations after the current iteration of LNS. We highlight a subset of 5 obstalces (red circles) and 5 waypoints (dark blue squares) that appear most frequently in the first neighborhood of the current decomposition. Qualitatively, the top 5 obstacles define some important junctions for waypoint updates. For waypoint updates, the highlighted ones tend to have large changes between iterations. Thus, a good decomposition focuses on important decision regions and allows for large updates in these regions.

B.3 Model Architecture

We first apply PCA to reduce the adjacency matrix obtained in Section 5.3. Then a fully-connected neural network is used to perform the classification task. Table B.8 lists the specifications. For MVC problems, for instance, we first apply PCA to reduce the adjacency matrix to 99 dimensions. Then the current solution assignments for each vertex is appended as described in Section 5.3, resulting in a 100-dimensional feature representation for each vertex. Next, it is passed through a hidden layer of 300 units with ReLU activations followed by a 2-class Softmax activations (since the model performs classifications). The number of classes is decided via the hyperparameter search for k as described in Section 5.4 and B.1.

B.4 Domain Heuristics

MVC. We compare with a 2-OPT heuristic based on local-ratio approximation (Bar-Yehuda and Even, 1983).

MAXCUT We compare with 3 heuristics. The first is the greedy algorithm that iteratively moves vertices from one cut set to the other based on whether such a

	PCA dimensions	Neural Network Architecture	Activation Functions
MVC	99	(100, 300, 2)	(ReLU, Softmax)
MAXCUT	299	(300, 100, 5)	(ReLU, Softmax)
CATS 2000	99	(100, 300, 100, 2)	(ReLU, ReLU, Softmax)
CATS 4000	399	(400, 300, 2)	(ReLU, Softmax)
Path Planning	499	(500, 100, 3)	(ReLU, Softmax)

Table B.8: Model architectures for all the experiments.

movement can increase the total edge weights. The second, proposed in Burer, Monteiro, and Y. Zhang, 2002, is based on a rank-two relaxation of an SDP. The third is from Sousa, Haxhimusa, and Kropatsch, 2013.

CATS We consider 2 heuristics. The first is greedy: at each step, we accept the highest bid among the remaining bids, remove its desired items and eliminate other bids that desire any of the removed items. The second is based on LP rounding: we first solve the LP relaxation of the ILP formulation of a combinatorial auction problem, and then we move from the bid having the largest fractional value in the LP solution down and remove items/bids in the same manner.

Appendix C

APPENDIX FOR CHAPTER 6

C.1 Proof for section 6.7

Proof of Theorem 9

Proof. The high-level idea is to first connect the total expected utility of the learned policy $\hat{\pi}$ with the expected utility of the expert policy π^{exp}, following the analysis in DAgger (Stéphane Ross, Gordon, and D. Bagnell, 2011). Then, we will use the fact that π^{exp} is greedy with respect to f, an approximation to the submodular utility function u, to bound the one step gain of the π^{exp} against the k step gain of running the optimal policy, and subsequently bound the total utility of the expert policy against the optimal policy. We would eventually obtain a similar result as Theorem 2, detailed as follows.

More concretely, following Theorem 3.4 in DAgger, we obtain that

$$\mathbb{E}[u(S_{\hat{\pi},k})] \geq \mathbb{E}[u(S_{\pi^{\text{exp}},k})] - \Delta_{\max} k \hat{\epsilon}_N - O(1).$$

Here $\Delta_{\max}$ is the largest one-step deviation from π^{exp} that $\hat{\pi}$ can suffer. It is equivalent to the term u in the DAgger paper. Since f is ϵ-close to a monotone submodular function u, we know that $\Delta_{\max} \leq \max_{A \subset \mathcal{V}, |A|=k} f(A) \leq \max_{A \subset \mathcal{V}, |A|=k} u(A) + \epsilon_E$, which is a constant once u is given.

Next, since π^{exp} is greedily optimizing an ϵ_E-approximation to a monotone submodular function u, we know that

$$\mathbb{E}[u(S_{\pi^{\text{exp}},k})] \geq (1 - 1/e)\mathbb{E}[u(S_{\pi^*,k})] - k\epsilon_E$$

following the proof from Theorem 5 in (Y. Chen, Renders, et al., 2017).

Combining both steps, we have that

$$\mathbb{E}[u(S_{\hat{\pi},k})] \geq (1 - 1/e)\mathbb{E}[u(S_{\pi^*,k})] - k(\epsilon_E + \Delta_{\max}\hat{\epsilon}_N) - O(1)$$

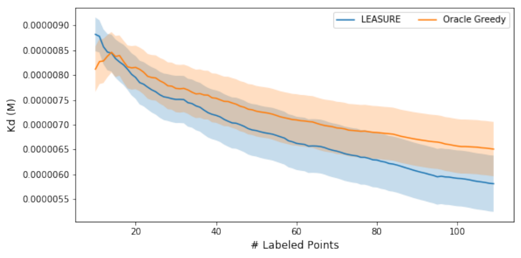
which completes the proof.

□

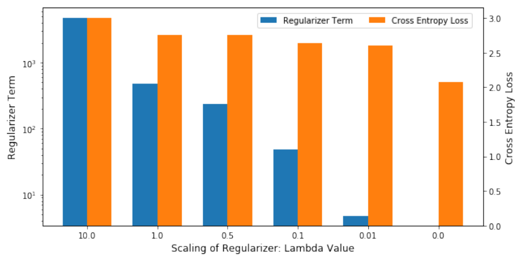
C.2 Supplemental Details for the Protein Engineering Experiments

Dataset Our datasets were identified in Protabank (C. Y. Wang et al., 2019) for training of active learning policies and benchmarking of performance. In selecting datasets upon which to train our active learning models several factors were considered. As the state space of possible protein variants for typical engineering application is very large, size is our foremost criteria. Additionally it will be advantageous to use datasets which characterize mutations to all amino acids (as opposed to Alanine scans), and those which include epistatic interactions. We also desire to identify datasets which have a high quality, quantitative readout, such as calorimetry, fluorescence, or SPR data.

Protein Engineering Methods Embeddings of protein sequences were created using the TAPE repository (Rao et al., 2019) according to the UniRep system as first proposed in Alley et al., 2019. UniRep produces protein embeddings as a matrix of shape (length protein sequence, 1900), although we average together the embeddings only of positions being engineered to produce a consistent embedding of shape (1900,). We have implemented the active learning imitation learning algorithm proposed in Liu, Buntine, and Haffari, 2018 to work with the protein embedding representations described above. Pseudocode for this method is presented in Algorithms 1 and 2 from the original work. As in Liu, Buntine, and Haffari, 2018, our policy network consists of a single dense unit which acts sequentially on the pool of samples being considered to produce a preference score. Our downstream protein engineering network (which was used to compute the preference score of the expert policy) acts on the protein embeddings prepared using TAPE. The network consists of an attention layer, followed by a 1-dimensional convolution layer (128 filters, kernel size 3), before being flattened and applying two fully connected layers of 128 units each. When predicting protein fitness, dropout is applied with a probability of 0.5 and an additional dense layer is applied with one unit and linear activation. Both networks are trained using ADAM with a learning rate of 1e-3. The implementation of this part of the project is nearly identical to Liu, Buntine, and Haffari, 2018, only changing the data representation, protein fitness network structure, and values of K (30), B (100) and T (20) as listed in the appendix of our work. Beta is fixed at 0.5, although the method was shown to be robust to a range of values. At training time, 100 labels are randomly selected for evaluating the effect of the greedy oracle, and 10 data are randomly selected to form the initial data set for learning. The superset is appended at each step of training the policy to maintain a size of 2x the labeled dataset. The training of a policy using these settings takes 36 hours on a modern multiprocessor computer equipped with an NVIDIA Titan V GPU.



(a) Comparison of policy to greedy oracle which it emulates



(b) Effect of scaling parameter lambda and empirical evidence for selecting its value

Figure C.1: Supplemental results for the protein engineering experiments of Section 6.8: (a) We observe that the policy learned by LEASURE preforms approximately as well as the greedy oracle which it emulates. In this experiment the policy was derived from the training set, but the greedy oracle is operating on the test set. (b) Lambda linearly scales the value of the regularizer term. When lambda takes value 0.01, the magnitude of the (scaled) regularizer term (represented by the blue bar) aligns the best with the magnitude of the cross entropy loss (represented by the orange bar). This is consistent with what we observed in Figure 6.3b where $\lambda = 0.01$ leads to well-regularized model behavior.