
A General Large Neighborhood Search Framework for Solving Integer Programs

Jialin Song[†] Ravi Lanka[‡] Yisong Yue[†] Bistra Dilkina^{*}
[†] California Institute of Technology
[‡]Jet Propulsion Laboratory, California Institute of Technology
^{*} University of Southern California

Abstract

This paper studies how to design abstractions of large-scale combinatorial optimization problems that can leverage existing state-of-the-art solvers in general purpose ways, and that are amenable to data-driven design. The goal is to arrive at new approaches that can reliably outperform existing solvers in wall-clock time. We focus on solving integer programs, and ground our approach in 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 approaches and their software implementations. We also show that one can learn a good neighborhood selector from training data. Through an extensive empirical validation, we demonstrate that our LNS framework can significantly outperform, in wall-clock time, compared to state-of-the-art commercial solvers such as Gurobi.

1 Introduction

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 inference in graphical models (Wainwright et al., 2005), planning (Ono and Williams, 2008), mechanism design (De Vries and Vohra, 2003), 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 these hard combinatorial optimization problems has spurred intense research in designing approximation algorithms and heuristics for large classes of combinatorial optimization settings, such as integer programming (Berthold, 2006; Fischetti and Lodi, 2010; Land and Doig, 2010) and satisfiability (Zhang and Malik, 2002; De Moura and Bjørner, 2008; Dilkina et al., 2009a). 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 from training data. The most popular paradigm, also referred to as “learning to search”, aims to learn good local decisions within a search procedure such as branch-and-bound (He et al., 2014; Khalil et al., 2016, 2017b; Song et al., 2018, 2019; 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. A major reason is that most algorithms are implemented on open-sourced solvers such as SCIP, which, according to recent benchmark results (Mittelmann, 2017; Optimization, 2019), is considerably slower than leading commercial solvers such as Gurobi and CPLEX (usually by a factor of 10 or more). Such learning to search approaches also ignore the many other heuristics typically employed by commercial solvers, such as primal pre-solve heuristics (Achterberg et al., 2019).

Motivated by the aforementioned drawback, in this paper, 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. We are further interested in designing

frameworks that are amenable to data-driven methods. We ground our work in two ways. First, we study how to solve integer programs (IPs), which are a common way to represent many combinatorial optimization problems. Second, we leverage the large neighborhood search (LNS) paradigm, which 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. We are furthermore interested in designing a framework that does not require incorporating extensive domain knowledge in order to apply to various problem domains, e.g., by learning data-driven decision procedures for the framework.

Our contributions can be summarized as:

- We propose a general LNS framework for solving large-scale IPs. Our framework does not depend on incorporating domain knowledge in order to achieve strong performance. In our experiments, we combine our framework with Gurobi, which is a leading commercial IP solver.
- We show that, perhaps surprisingly, even using a *random* decision procedure within our LNS framework finds significantly outperforms Gurobi on many problem instances.
- We develop a learning-based approach that predicts a partitioning of the variables of an IP, which then serves as a learned decision procedure within our LNS framework. In a sense, this data-driven procedure is effectively learning how to decompose the original optimization problem into a series of smaller sub-problems that can be solved much more efficiently using existing solvers.
- We perform an extensive empirical validation across several IP 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.

2 Related Work on Learning to Optimize

An increasingly popular paradigm for the automated design and tuning of solvers is to use data-driven or 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; and (3) learning to identify key substructures that an existing solver can exploit, such as backdoor variables. In this section, we survey these three paradigms.

2.1 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 (IPs) 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 (Khalil et al., 2016; Gasse et al., 2019), or node selection (He et al., 2014; Song et al., 2018, 2019). Moreover, one can also train a model to decide when to run primal heuristics endowed in many IP solvers (Khalil et al., 2017b). 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 et al., 2014; Khalil 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 IP 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.

Beyond branch-and-bound, other search frameworks that are amenable to data-driven design include A* search (Song et al., 2018), direct forward search (Khalil et al., 2017a), and sampling-based planning (Chen et al., 2020). These settings are less directly relevant, since our work is grounded in solving IPs. However, the LNS framework can, in principle, be interpreted more generally to include these other settings as well, which is an interesting direction for future work.

2.2 Algorithm Configuration

Another area of using learning to speed up optimization solvers is algorithm configuration (Hoos, 2011; Hutter et al., 2011; Ansótegui et al., 2015; Balcan et al., 2018; Kleinberg et al., 2019). Existing solvers tend to have many customizable hyperparameters whose val-

ues strongly influence the solver behaviors. Algorithm configuration aims to optimize those parameters on a problem-by-problem basis to speed up the solver.

Similar to our approach, algorithm configuration approaches leverage existing solvers. One key conceptual difference is that algorithm configuration does not yield fundamentally new approaches, but rather is a process for tuning the hyperparameters of an existing approach. As a consequence, one limitation of algorithm configuration approaches is that they rely on the underlying solver being able to solve problem instances in a reasonable amount of time, which may not be possible for hard problem instances. Our LNS framework can thus be viewed as a complementary paradigm for leveraging existing solvers. In fact, in our experiments, we perform a simple version of algorithm configuration. We defer incorporating more complex algorithm configuration procedures as future work.

2.3 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 (Dilkina et al., 2009b), which are a set of variables that, once instantiated, the remaining problem simplifies to a tractable form (Dilkina et al., 2009a). Our approach bears some high-level affinity to this paradigm, as we effectively aim to learn decompositions of the original problem into a series of smaller subproblems. However, our approach makes a much weaker structural assumption, and thus can more readily leverage a broader suite of existing solvers. 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).

3 A General Large Neighborhood Search Framework for Integer Programs

We now present our large neighborhood search (LNS) framework for solving integer programs (IPs). LNS is a metaheuristic that generalizes the neighborhood search for optimization which 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 the vehicle routing problem, 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 4. In particular, our approach can be viewed as a decomposition-based LNS framework that operates on generic IP representations, as described in Section 3.2.

3.1 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 et al., 2006) and its extension of k -opt operation (Helsgaun, 2009). LNS defines $N(s)$ implicitly through a *destroy* and a *repair* method. A destroy method deconstructs 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 IPs, the LNS is also used as a primal 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 bandit algorithm selection from a pre-defined set (Hendel, 2018). Furthermore, because of the level of decision-making, 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; 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 IPs. 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.

3.2 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 integer program (IP) via defining decompositions of its integer variables into disjoint subsets. Afterwards, we can se-

Algorithm 1 Decomposition-based LNS

- 1: **Input:** an optimization problem P , an initial solution S_X , a decomposition $X = X_1 \cup X_2 \cup \dots \cup X_k$, a solver F
- 2: **for** $i = 1, \dots, k$ **do**
- 3: $S_X = \text{FIX_AND_OPTIMIZE}(P, S_X, X_i, F)$
- 4: **end for**
- 5: **return** S_X

lect 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 IP instantiation of any combinatorial optimization problem.

Throughout this paper, we consider minimization of the objective value for all the problems. We first describe a version of LNS for integer programs based on decompositions of their integer variables which is a modified version of the evolutionary approach proposed in Rothberg (2007). The algorithm is outlined in Alg 1.

For an integer program P with a set of integer variables X (not necessarily all the integer variables), we define a decomposition of the set X as a disjoint union $X_1 \cup X_2 \cup \dots \cup X_k$. 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 1). As the resulting optimization is a smaller IP, we can use any off-the-shelf IP solver to carry out the local search. In our experiments, we use Gurobi to optimize the sub-IP. 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 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 4.

4 Learning a Decomposition

In this study, we apply data-driven methods, such as reinforcement learning and imitation learning, to learn policies to generate decompositions for the LNS framework described in Section 3.2. We specialize a Markov decision process for our setting. For a combinatorial opti-

Algorithm 2 COLLECT_DEMOS

- 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 decomposition, F a solver.
- 2: **for** $i = 1, \dots, n$ **do**
- 3: $best_obj \leftarrow \infty$
- 4: $best_decomp \leftarrow None$
- 5: **for** $j = 1, \dots, m$ **do**
- 6: $decomps \leftarrow []$
- 7: **for** $t = 1, \dots, T$ **do**
- 8: $X \leftarrow \text{RANDOM_DECOMPOSITION}(k)$
- 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** $best_decompos$

mization 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 $a \in \mathcal{A}$ at a state s is a decomposition of X as described in Section 3.2. 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 ourselves to finite-horizon task of length T so we can set the discount factor γ to be 1.

4.1 Reinforcement Learning

For reinforcement learning, for simplicity, we choose to use REINFORCE (Sutton et al., 2000) which is a classical Monte-Carlo policy gradient method for optimizing policies. To 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

$$\nabla_\theta \eta(\pi_\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) \sum_{t'=t}^T r(s_{t'}, a_{t'}) \right].$$

By sampling trajectories $(s_0, a_0, \dots, s_{T-1}, a_{T-1}, s_T)$, one can estimate the gradient $\nabla_\theta \eta(\pi_\theta)$.

4.2 Imitation Learning

In imitation learning, demonstrations (from an expert)

Algorithm 3 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 decomposition,  $F$  a solver.
2: for  $t = 1, \dots, T$  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  $i = 1, \dots, n$  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  $\pi_1, \pi_2, \dots, \pi_T$ 
```

serves as the learning signals. However, we do not have the access to an expert to generate good decompositions. To overcome this issue, we generate demonstrations by sampling random decompositions and take the ones resulting in best objectives as demonstrations. This procedure is shown in Alg 2. 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 1) 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). The main idea is to turn each demonstration trajectory $D_i = (s_0, a_0, s_1, a_1, \dots, s_{T-1}, a_{T-1})$ into a collection of state-action pairs $\{(s_0, a_0), (s_1, a_1), \dots, (s_{T-1}, a_{T-1})\}$, then treat policy 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 has a label about which subset X_i it belongs to. Thus, we reduce the learning problem to a supervised classification task.

Behavior cloning suffers from cascading errors (Ross and Bagnell, 2010). We use the forward training algorithm (Ross and Bagnell, 2010) to correct mistakes made at each step. We adapt the forward training algorithm for our use case and present it in Alg 3 that uses Alg 2 as a subroutine. 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.

4.3 Featurization of an Optimization Problem

For training models, it is necessary to define features that contain enough information for learning. In the fol-

lowing paragraphs, 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 Khalil et al. (2017a). 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. Notice that for such optimization problems, each vertex in the graph is often associated with an integer variable in its IP formulation.

General Integer Programs. There are other classes of combinatorial optimization problems that do not originate from explicit graphs. Nevertheless, they can be modeled as integer 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 3.2, 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. As a result, we can simply include the variable value in the solution as an additional feature.

5 Empirical Validation

We present experimental results on four diverse applications covering both combinatorial optimization over graphs and general IPs. We discuss the design choices of crucial parameters in Section 5.1, and present the main results in Sections 5.2 & 5.3. Finally, we inspect visualizations to interpret predicted decompositions in Section 5.4, and discuss some limitations in Section 5.5.

5.1 Datasets & Setup

Datasets. We evaluate on 4 NP-hard benchmark problems expressed as IPs. 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

$k \backslash t$	1	2	3
2	13.61 ± 0.82	14.19 ± 0.89	14.42 ± 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 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.

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 et al., 2000) and risk-aware path planning (Ono and Williams, 2008), which are not based on graphs. We use the Combinatorial Auction Test Suite (CATS) (Leyton-Brown et al., 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 procedure 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 in order to run the our LNS approach. The first one is k , the number of equally sized subsets to divide variables X into. The second is how long we run the solver on each sub-problem. Each sub-IP could still be fairly large so solving them 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-IP 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 $\frac{\Delta}{T}$ where Δ is the objective value

improvement and T is the time spent as the selection criterion for the optimal configuration. Table 1 contains the result for one MVC dataset. As shown, the configuration makes a big difference on the performance. For this case, $(k, t) = (2, 3)$ is the best setting and we will use them for our experiments on this particular dataset. We perform the procedure for every dataset. See Appendix for similar tables for other datasets.

5.2 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

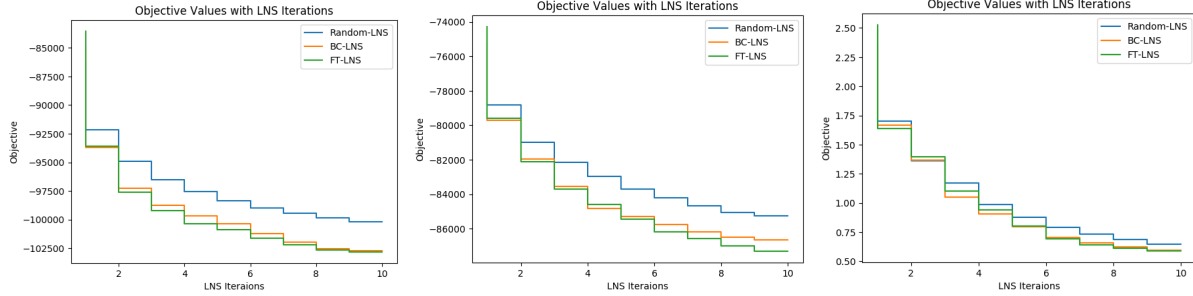
We use Gurobi 9.0 as the underlying solver. For learned LNS methods, we generate 10 decompositions in sequence by the model and apply LNS with these decompositions. We use the same time limit setting for running each sub-problem, as a result, the wall-clock among decomposition methods are very close.

When comparing using just Gurobi, we limit Gurobi’s runtime to be 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 more time to find the best solution possible.

Main Results. Tables 2, 3 and 4 show the results across the different benchmarks. We make two observations:

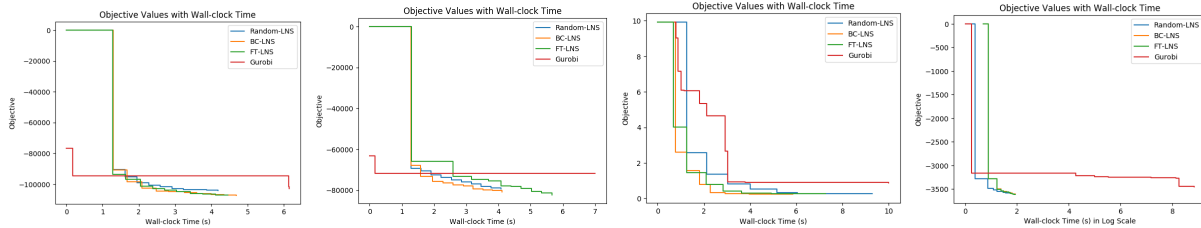
- All LNS variants significantly outperform Gurobi (up to 50% improvement in objectives), given the same amount or less wall-clock time. Perhaps surprisingly, this phenomenon holds true even for Random-LNS.
- The imitation learning based variants, FT-LNS and BC-LNS, outperform Random-LNS and RL-LNS in 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 5.4. It is possible that a more sophisticated RL method could further improve RL-LNS.



(a) Combinatorial auction of 2000 items and 4000 bids from the regions distribution. (b) Combinatorial auction of 2000 items and 4000 bids from the arbitrary distribution. (c) Risk-aware path planning for navigating through 30 obstacles.

Figure 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) Combinatorial auction of 2000 items and 4000 bids from regions distribution. (b) Combinatorial auction of 2000 items and 4000 bids from arbitrary distribution. (c) Risk-aware path planning for navigating through 30 obstacles. (d) Maximum cut over a Barabási-Albert random graph with 500 vertices.

Figure 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 the advantage is maintained even though Gurobi is given more time for each instance. In Fig 2d, after running for 2 hours, Gurobi is unable to match the quality of solution found by Random-LNS in 5 seconds.

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. Fig 1 shows objective value progressions of variants of our LNS approach on three datasets. For 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. Figure 2 shows four representative instances. We see that BC-LNS achieves the best performance profile of solution quality vs. wall-clock.

How Long Does Gurobi Need? Figure 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 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).

5.3 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. 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.

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

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

Table 2: Comparison of different LNS methods and Gurobi for MVC and MAXCUT problems with different random graph generators.

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

Table 3: Comparison of different LNS methods and Gurobi for CATS problems with different bid distributions.

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

Table 4: Comparison of learning-based methods with the random method and Gurobi for the risk-aware path planning problems with different number of obstacles.

	MVC BA 1000	MVC ER 1000
Local-ratio	487.58 \pm 1.16	498.20 \pm 1.24
Best-LNS	432.00 \pm 0.52	470.04 \pm 0.37

Table 5: Comparison between LNS with the local-ratio heuristic for MVC.

Table 5 summarizes the results. The best LNS result outperforms by 11% on BA graphs and 6% on ER graphs.

For 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 movement can increase the total edge weights. The second, proposed in Burer et al. (2002), is based on a rank-two relaxation of an SDP. The third is from de Sousa et al. (2013). The results are presented in Table 6. The SDP-based heuristic performs best for both random graph distributions, which shows that a specially designed heuristic can still outperform a general IP solver.

For 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 IP 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. As shown in Table

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

Table 6: Comparison between LNS with three heuristics for MAXCUT.

7, Our LNS approach outperforms both methods by up to 50% in objective values.

5.4 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 with 20 obstacles for a clearer view. Binary variables in an IP 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 3 captures 4 consecutive iterations of LNS with large solution improvements. 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 obstacles (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.

	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	-102247.9 ± 709.0	-208586.3 ± 1211.7	-87311.8 ± 676.0	-172268.1 ± 1594.8

Table 7: Comparison between LNS with greedy and LP rounding heuristics for combinatorial auctions.

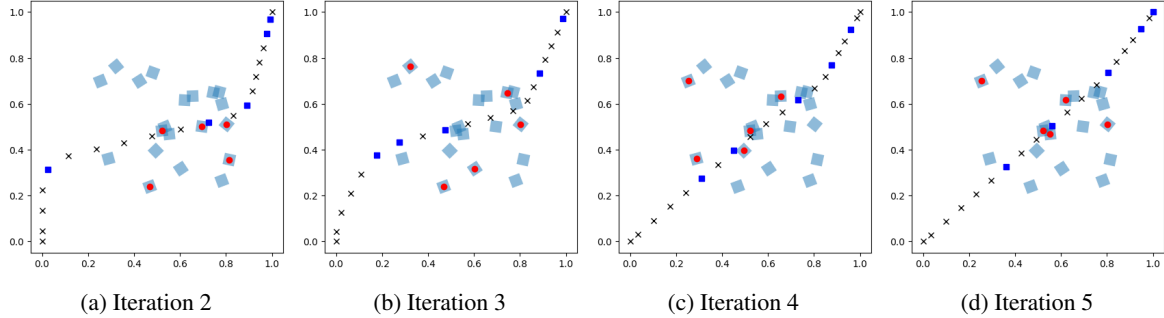


Figure 3: Visualizing predicted decompositions in a risk-aware path planning problem, with 4 consecutive solutions after 3 iterations of LNS. Each blue square is an obstacle and each cross is a waypoint. The obstacles in red and waypoints in dark blue are the most frequent ones in the subsets that lead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

5.5 Limitations

While we have presented positive results on our proposed general LNS method, limitations exist. A major one is that neighborhoods need to be large for LNS to work well. For example, if we compare Random-LNS with Gurobi under the same wall-clock time limit on a combinatorial auction dataset with only 500 items and 1000 bids. Gurobi outperforms Random-LNS on the objective value for about 3% (-26986.72 vs -26156.72). This means that LNS is most effective in dealing with large scale problem instances, not necessarily for small ones. As a result, it is important to evaluate whether a problem is of the scale where the proposed LNS method is useful.

6 Conclusion & Future Work

We have presented a general large neighborhood search framework for solving integer programs. Our extensive benchmarks show the proposed method consistently outperform Gurobi in wall-clock time across diverse applications. Our method is also competitive with strong heuristics in specific problem domains.

We believe our current research has many exciting future directions that connect different aspects of the learning to optimize research. Our framework relies on a good solver thus the progress from the learning to search community can lead to better LNS results. We have briefly mentioned a version of the algorithm configuration problem we face in Section 5.1, but the full version that adaptively chooses the number of subsets and time limit is surely more powerful. Algorithm configuration can also

help optimize the hyperparameters of a solver for solving each sub-problem. Finally, our approach is closely tied to the effort to identify substructures in optimization problems. A deeper understanding of them can inform the design our data-driven methods and define new variants of the learning problem within the proposed framework, e.g., learning to identify backdoor variables.

References

- T. Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 2009.
- T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 2019.
- R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.
- R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.
- C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, and K. Tierney. Model-based genetic algorithms for algorithm configuration. In *AISTATS*, 2015.
- M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to branch. In *ICML*, 2018.
- R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. Technical report, Computer Science Department, Technion, 1983.
- B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. 2013.
- T. Berthold. Primal heuristics for mixed integer programs. 2006.

- C. Blikelú, P. Bonami, and A. Lodi. Solving mixed-integer quadratic programming problems with ibm-cplex: a progress report. In *RAMP*, 2014.
- S. Burer, R. D. Monteiro, and Y. Zhang. Rank-two relaxation heuristics for max-cut and other binary quadratic programs. *SIAM Journal on Optimization*, 12(2):503–521, 2002.
- B. Chen, B. Dai, and L. Song. Learning to plan via neural exploration-exploitation trees. In *ICLR*, 2020.
- J. Cui, Y. S. Chu, O. O. Famodu, Y. Furuya, J. Hattrick-Simpers, R. D. James, A. Ludwig, S. Thienhaus, M. Wuttig, Z. Zhang, et al. Combinatorial search of thermoelastic shape-memory alloys with extremely small hysteresis width. *Nature materials*, 2006.
- L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.
- S. de Sousa, Y. Haxhimusa, and W. G. Kropatsch. Estimation of distribution algorithm for the max-cut problem. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 244–253. Springer, 2013.
- S. De Vries and R. V. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on computing*, 2003.
- B. Dilkina, C. P. Gomes, Y. Malitsky, A. Sabharwal, and M. Sellmann. Backdoors to combinatorial optimization: Feasibility and optimality. In *CPAIOR*, 2009a.
- B. Dilkina, C. P. Gomes, and A. Sabharwal. Backdoors in the context of learning. In *SAT*, 2009b.
- M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.
- P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 1960.
- M. Fischetti and A. Lodi. Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science*, 2010.
- M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *NeurIPS*, 2019.
- H. He, H. Daume III, and J. M. Eisner. Learning to search in branch and bound algorithms. In *NeurIPS*, 2014.
- S. Helber and F. Sahling. A fix-and-optimize approach for the multi-level capacitated lot sizing problem. *International Journal of Production Economics*, 123(2):247–256, 2010.
- K. Helsgaun. General k-opt submoves for the lin-kernighan tsp heuristic. *Mathematical Programming Computation*, 1(2-3): 119–163, 2009.
- G. Hendel. Adaptive large neighborhood search for mixed integer programming. 2018.
- H. H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. Springer, 2011.
- A. Hottung and K. Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. *arXiv:1911.09539*, 2019.
- F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION*, 2011.
- E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *NeurIPS*, 2017a.
- E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *AAAI*, 2016.
- E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *IJCAI*, 2017b.
- Y. Kim, S. Wiseman, A. C. Miller, D. Sontag, and A. M. Rush. Semi-amortized variational autoencoders. In *ICML*, 2018.
- R. Kleinberg, K. Leyton-Brown, B. Lucier, and D. Graham. Procrastinating with confidence: Near-optimal, anytime, adaptive algorithm configuration. In *NeurIPS*, 2019.
- A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.
- K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *EC*, 2000.
- Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.
- A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. In *ICML*, 2017.
- H. D. Mittelmann. Latest benchmarks of optimization software. 2017.
- M. Ono and B. C. Williams. An efficient motion planning algorithm for stochastic dynamic systems with constraints on probability of failure. In *AAAI*, 2008.
- G. Optimization. Gurobi 8 performance benchmarks, 2019. <https://www.gurobi.com/pdfs/benchmarks.pdf>.
- D. Pisinger and S. Ropke. Large neighborhood search. *Handbook of metaheuristics*, 2010.
- D. A. Pomerleau. Alvin: An autonomous land vehicle in a neural network. In *NeurIPS*, 1989.
- S. Ross and D. Bagnell. Efficient reductions for imitation learning. In *AISTATS*, 2010.
- E. Rothberg. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, 19(4):534–541, 2007.
- P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP*, 1998.
- J. Song, R. Lanka, A. Zhao, Y. Yue, and M. Ono. Learning to search via retrospective imitation. *arXiv:1804.00846*, 2018.
- J. Song, R. Lanka, Y. Yue, and M. Ono. Co-training for policy learning. In *UAI*, 2019.
- R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS*, 2000.
- A. A. Syed, K. Akhnoukh, B. Kaltenhaeuser, and K. Bogenberger. Neural network based large neighborhood search algorithm for ride hailing services. In *EPIA*, 2019.
- M. Wainwright, T. Jaakkola, and A. Willsky. Map estimation via agreement on trees: message-passing and linear programming. *IEEE transactions on information theory*, 2005.
- L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CAV*, 2002.

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

Table 8: Parameter sweep results for (k, t) of the MVC dataset for Barabási-Albert random graphs with 1000 vertices.

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

Table 9: Parameter sweep results for (k, t) of the MAXCUT dataset for Erdős-Rényi random graphs with 500 vertices.

A Appendix

A.1 More Algorithm Configuration Results

In this section, we present the more algorithm configuration results similar to the one in Section 5.1.

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

Table 10: Parameter sweep results for (k, t) of the MAXCUT dataset for Barabási-Albert random graphs with 500 vertices.

$k \backslash t$	1	2	3
2	65360.28 ± 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 11: Parameter sweep results for (k, t) of the CATS dataset for the regions distribution with 2000 items and 4000 bids.

$k \backslash t$	1	2	3
2	54358.95 ± 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 12: Parameter sweep results for (k, t) of the CATS dataset for the arbitrary distribution with 2000 items and 4000 bids.

$k \backslash t$	1	2	3
2	0.37 ± 0.18	0.39 ± 0.07	0.36 ± 0.05
3	0.41 ± 0.07	0.43 ± 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 13: Parameter sweep results for (k, t) of the risk-aware path planning for 30 obstacles.