CS159 Project Report: An Exercise in Structured LNS and Reinforcement Learning

Ivan Dario Jimenez **Bryan Yao** **Jessica Choi** **Serena Yan**

1 Introduction

Optimization-based planners have become pervasive in the area robotic locomotion. From Unmanned Aerial vehicles (UAVs) [3] to Bipeds [2], optimization-based planning has shown powerful results in its own right.

A mixed-integer optimization program can be used to plan a path for a robot that avoids obstacles. Notice that due to the discrete nature of certain constraints such as collisions, discrete variables are an essential tool for the purposes of planning. In general, the planning problem is non-convex; however, by using our understanding of the structure of the problem, we can partition the traversable space into a set of convex spaces. These spaces can in turn be used in a Mixed-Integer Optimization (MIP) problem. These types of control and planning techniques rely solely on the structure of the problem to generate a very general policy. Unfortunately they require a significant amount of tuning for specific environments and can be computationally expensive (in the order of minutes) for even simple environments. Unfortunately, this means MIP-based policies are only feasible for trajectory optimization rather than real-time applications like MPC.

Meanwhile in the machine learning community, reinforcement learning has proven to be extremely successful at similarly complex simulated robotic tasks, such as having robots mimic human movement or grasp objects with visual feedback [7][6]. Even more generally, there have been outstanding results in games with discrete action spaces such as Go and StarCraft[1][11]. Sadly, these methods based solely on reinforcement learning can pose serious risks to robots during training in dynamic environments and require unrealistic amounts of data.

This project aims to use machine learning to reduce the computational complexity of path planning MIP's. This could allow us to use more reliable policies while keeping the overall computational cost reasonable for real-time applications.

2 Background

Overall, this project follows the framework of learning to optimize. In it we attempt to augment an optimization solver using data to improve performance. Consider for example augmentation of Branch & Bound as presented in [4] where the authors attempt to learn the branching heuristic to speed up solving. Unfortunately, this approach requires direct access to the inside components of optimization solvers. Unless there is built in support this algorithm, it cannot be implemented using commercial solvers such as Gurobi. This is a problem particularly if performance-conscious applications where the many performance improvements these solvers implement are a requirement.

There are also alternative approaches that exploit the inherent graph structure of some classes of optimization problems to generate algorithmic optimization as shown in [5]. Although part of the planning problem is often interpreted as a discrete optimization problem on graphs, this approach would fail to take into account he continuous dynamics of a robotic system.

Instead, we focused on working with the Large Neighborhood Search (LNS) framework as presented in [12] where the authors use method that the discrete variables of an optimization problem to significantly reduce the computational cost while remaining close to optimal.

Preprint. Under review.

3 Research question and approach

With this project we want to understand how to use LNS and RL to speedup MIP planning for robot locomotion. To formally understand this research question we must first present motion planning in a dynamic context.

Consider a dynamical system defined by the differential equation

$$\dot{x} = f(x, u) \tag{1}$$

Where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$. Although we intended to use general dynamic systems like those presented in [2] and [3], unfortunately the code for these papers was deprecated and removed from Drake in a recent overhaul [13] as discussed in this issue [8]. Thus we will limit our exploration to double integrator linear dynamical system that can be represented as:

$$\dot{x} = Ax + Bu \tag{2}$$

Where $x \in \mathbb{R}^2$ and $u \in \mathbb{R}^2$. Notice that we are only controlling the accelerations of the system. Furthermore we have chosen A to some natural drift that will becomes important when dealing with min-norm controllers that avoid obstacles.

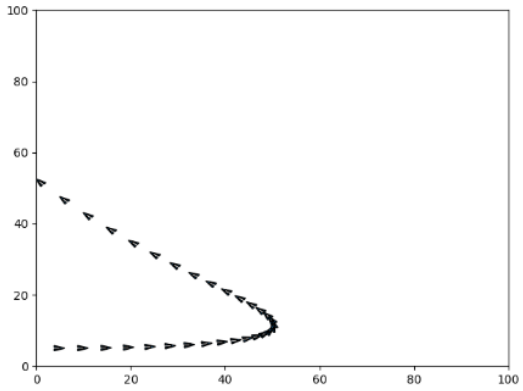


Figure 1: Unconstrained Dynamics Test

3.1 The Trajectory Optimization Problem:

Now that we have a set of defined dynamics we must define the trajectory optimization problem that will solve for an optimal path. The goal is to go from x_0 to x_f so that $\dot{x} = Ax + Bu$. Also, notice that $x \in \mathcal{S}$ where $\mathcal{S}$ a set of regions without obstacles. With this we can write a simple MIQCP (Mixed-Integer Quadratic program)

$$\min_{u_{1:T}} \sum_{t=1}^{T} u_t^\top R u_t \tag{3}$$

$$\text{s.t. } x_{t+1} - x_t = \frac{1}{2}(t_{k+1} - t_k)(f(x_{t+1}, u_{t+1}) + Ff(x_t, u_t)) \tag{4}$$

$$x_1 = x_0 \tag{5}$$

$$x_T = x_f \tag{6}$$

$$x_t \in \mathcal{S} \quad \forall 1 \leq t \leq T \tag{7}$$

This would normally be a regular quadratic program if not for the $x \in S$ constraint which introduces the discrete decision variable: $\mathcal{S} = \{x | \exists_{i \in \mathbb{N}} \quad A_i x \leq b_i\}$ This simply means that x is in a polytope

parametrized by A_i and b_i which correspond to an area without obstacles.Notice that T is the number of waypoints in our trajectory at which we evaluate the discrete dynamics. $t_{k+1} - t_k$ is the amount of continuous time that goes by between waypoints.

Notice here that at each waypoint we must decide which convex region it belongs to so that means that each waypoint must have a set of N discrete decision variables. This means that in a problem with T waypoints the algorithm will have a runtime of $\mathcal{O}(\in^{\mathcal{NT}})$. For a size of $T = 200$ and $N = 4$, for example we observed runtimes in the order of 10 seconds depending on the starting condition.

3.2 Our LNS implmentation

Throughout this paper we will be using a variant of decomposition-based LNS which can be summarize as follows:

Algorithm 1 Decomposition-based LNS

procedure LNS($P, x_{init}, X = X_1 \cup X_2 \cup \ldots \cup X_k, F$)
    $S_x \leftarrow x_{init}$
    for $i \in \{1 \ldots k\}$ **do**
        $S_X \leftarrow$FIX_AND_OPTIMIZE(P, S_x, X_i, F)
    end for
end procedure

In algorithm 1 X, is a set of disjoint subsets of variables to optimize over, x_{init} is an initial solution to the solver, P is the optimization problem and F is the solver. The complexity of LNS comes in how we choose the disjoint subsets X and the value that is assigned to the variables that are not part of the disjoint subset which is, included in S_x.

3.2.1 Purely Randomized LNS

At the beginning we attempted a purely random approach to selecting the disjoint subsets X. Furthermore for each $X_i \in X$ we must have an assignment $\Gamma_x \subset S_x$ for the variables that we have not being optimized. Notice, however that the dynamics constraints enforces a form of continuity constraint on the path the robot must follow. Thus, it makes no sense to, for example, assign adjacent waypoints x_t and x_{t+1} to different convex regions that are not overlapping. Using this approach we realized that we needed a way to make reasonable variable assignments that satisfied the dynamics constraint without actually solving the problem. We implemented two separate approaches that tackled this issue: Reinforcement learning and Randomized path assignment.

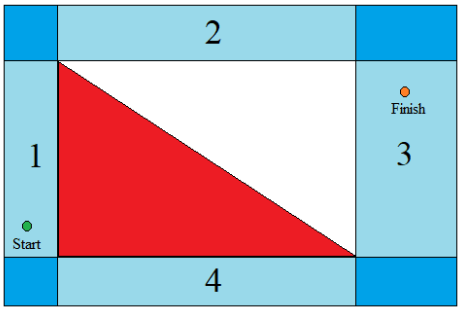
3.3 Graph Structure of MIC

Clearly, we do not have a way to verify that a variable assignment will satisfy the dynamics constraints without solving the optimization problem. Instead we will use a simple heuristic: if two regions R_i and R_l overlap at at any point, there exists sequence of control inputs u_t^{ab} that can take the system from $a \in R_i$ to $b \in R_l$.

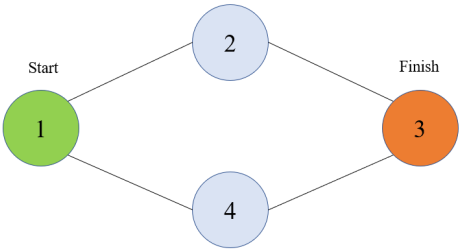
Using this assumption we can reduce the optimization problem to a graph path-finding problem as shown in fig. 2. Notice that the resulting graph is undirected and all edges have the same weight. This is clearly not true in practice but with this heuristic we can create more feasible assignments than with the purely randomized approach. Dealing with the infeasible assignments that this approach leaves is handled differently between the randomized approach and the RL approach.

3.4 Graph Augmented Randomized LNS

For Graph Augmented Randomized LNS, described in algorithm 2, we loaded an environment (list of convex regions), and generated a graph connecting overlapping convex regions. From this graph, we generated all possible paths from the starting region to the ending region and randomly selected a path, $path$. This would ensure that our LNS would search through an actually feasible progression of regions for getting from the start to the finish. From our path we obtain a solution S_x, X. From this point, we split up our solving of the problem into an adjustable amount of time steps, where



(a) 2D Environment with obstacles in red and safe regions in blue. Notice that the safe regions overlap in the corners.



(b) In this graph every node represents a region and every edge connects two regions that overlap.

Figure 2: Here we can see how to convert the set of safe convex regions into a graph.

Algorithm 2 Using Graph for Set Generation for LNS

procedure GRAPHLNS($P, F, T, G, \Delta t$)
  $path \leftarrow$random_path(G)
  $S_x, X \leftarrow$ path(0)
  for $i \in \{1 \dots \frac{T}{\Delta t}\}$ **do**
    $S_{x\Delta t} \in S_x$
    $X_{x\Delta t} \in X$
    $S_x \leftarrow$LNS($P, S_{x\Delta t}, X_{x\Delta t}, F$)
  end for
  return $S_x x$
end procedure

T is the total time, and Δt is the length of each time step. At each time step we activate a subset of the solution $S_{x\Delta t}, X_{x\Delta t}$, and pass this into the LNS, which will update the solution S_x. We use this updated value and the next time step to determine the new active and inactive sets for the next iteration, until all time steps are complete.

3.5 Graph Augmented Reinforcement Learning

Consider the function $\pi : A, b, x_{init}, x_f \rightarrow X$ that maps from the set convex polytopes A, b, the starting location x_{init} and the final location x_f. We can attempt to estimate such a function using an the reinforcement learning paradigm where the state is the tuple $\mathcal{S} = S_x$ which encodes the state of LNS at the current iteration, has the action space $\mathcal{A} = \mathscr{X}$ where $\mathscr{X}$ is the set of all possible sequences of disjoint sets of discrete variables for the optimization problem P. The reward $\mathcal{R}(s, a) = \frac{C^* - \hat{C}}{C^*}$ where C^* is the cost of the optimization problem with all regions active and $\hat{C}$ is the cost for the current iteration of LNS using S_x and X . Here LNS is the variation shown in algorithm 2 but with π generating the sequence of disjoint subsets X_i rather than randomly selecting paths.

Notice that for a trajectory of T waypoints we are solely activating a region Δt for the optimizer to solve for. This window of variables is slid from the beginning of the trajectory to the end. Thus we can frame the optimization of an entire trajectory as an episode and each individual choice of X_i as a step. Using this framework we implemented reinforce to attempt to find a reasonable policy. To mitigate any numerical issues we clip the reward so that $\mathcal{R} \in [-1000, 0]$

Notice the filter box in fig. 3. Although it's not obvious how to enforce the hard adjacency constraints seen in the graph at any point in the trajectory there are three cases where we can enforce proximity constrains:

1. X_0: We know trivially that the first window must contain the initial location. So with this we always know the value of the discrete variable corresponding to the initial section of the

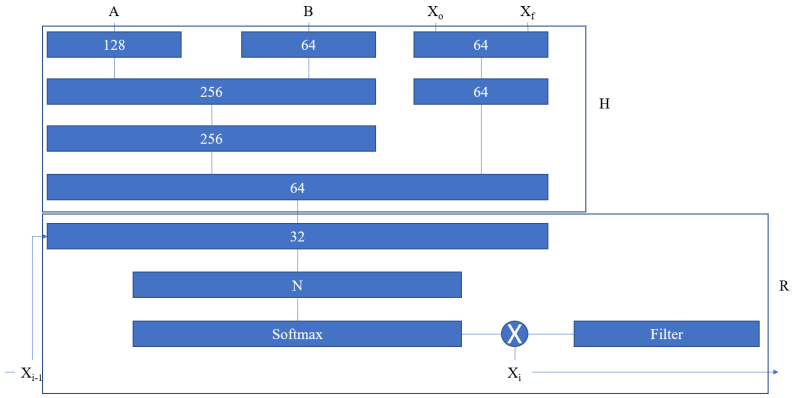


Figure 3: Architecture used to generate trajectory.

trajectory and can avoid optimizing over them. Thus we can restrict the possible values for X_1 to adjacent regions in G.

2. $X_{\frac{T}{\Delta t}}$: Similarly, as in the previous case, since we know the target location, we know the region for the last section of the trajectory. Thus we can restrict the possible values for X_{f-1} to adjacent regions in G
3. $\{X_i | i < \Delta t\}$: Suppose we have already optimized over several windows Δt, then the value of S_X for those discrete optimization variables will not change in future iterations. Thus we can restrict the values of $\Delta t + 1$ to regions adjacent to the current value of Δt

We implemented these regions using a filter that simply multiplied by zero the possible values for X_i that the network would normally give. Notice that this outputs a vector of probable regions which are then sampled from in order to carry the rest of the REINFORCE algorithm.

3.6 Proximal Policy Optimization (PPO)

Algorithm 3 Using PPO for optimizing across trajectories

procedure PPO(N, T, θ_0)
  for iteration $\in \{0, 1, 2, \ldots\}$ **do**
    $r(\theta) \leftarrow \dfrac{\pi_\theta(a|s)}{\pi_{\theta iter}(a|s)}$
    for actor $\in \{1, 2, \ldots, N\}$ **do**
      for $t \in \{1 \ldots T\}$ **do**
        Collect trajectories by running policy $\pi_{\theta iter}$
        Compute rewards $\hat{R}_t$ and advantage estimates $\hat{A}_t$
      end for
    end for
    $\theta_{iter+1} \leftarrow \arg\max_\theta \mathbb{E}[\min(r_t(\theta)\hat{A}_{\theta iter}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$
  end for
  return θ
end procedure

The Proximal Policy Optimization algorithm as described in [9] is an improvement on the more complicated Trust Region Policy Optimization (TRPO) algorithm. PPO converges much more quickly by using first-order methods instead of the complex second-order methods used by TRPO, and PPO also utilizes clipped probability ratios to generate a lower bound on the results, by limiting the amount that the policy can change from each update. Even though clipping the surrogate objective

significantly simplifies the algorithm and implementation, the performance is maintained when compared to TRPO.

3.7 Trust Region Policy Optimization (TRPO)

Algorithm 4 Using TRPO for optimizing across trajectories

procedure TRPO($\theta_0, \phi_0, \delta, \alpha, K$)
  for k = 0,1,2,... **do**
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$
    Compute rewards-to-go $\hat{R}_t$
    Compute advantage estimates $\hat{A}_t$ based on the current value function V_{ϕ_k}
    Estimate policy gradient as $\hat{g_k} = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t$
    Use the conjugate gradient algorithm to compute $\hat{x_k} \approx {\hat{H}_k}^{-1} \hat{g_k}$ where $\hat{H}_k$ is the Hessian of the sample average KL-divergence
    Update the policy by backtracking line search with $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x_k}^T \hat{H}_k \hat{x_k}}} \hat{x_k}$ where $j \in \{0, 1, 2, ..., K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint
    Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2$$

  end for
end procedure

The algorithm is taken from [10]. TRPO is a type of reinforcement learning that updates policies by taking the largest step possible to improve performance, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is expressed in terms of KL-Divergence, a measure of distance between probability distributions.

4 Experiments and Results

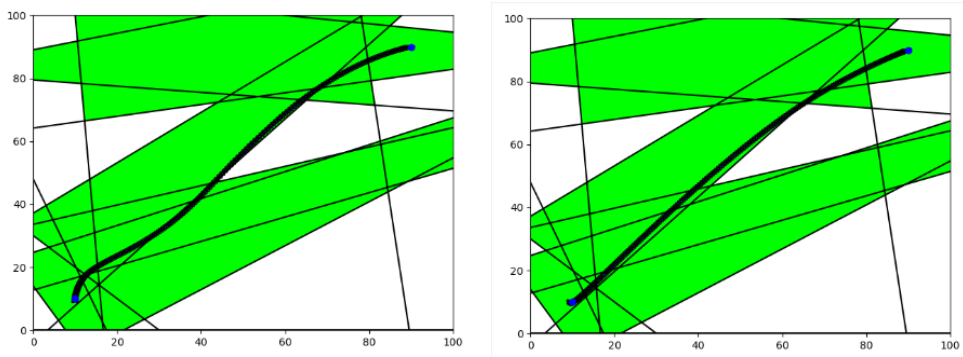
4.1 Graph Augmented Randomized LNS vs PPO vs TRPO vs Gurobi

Observe the following results from comparing the performance of Graph Augmented Randomized LNS, the optimal solution from Gurobi, and PPO. First, on an average environment:

Table 1: Objective and Runtime Comparison on Average Environment

Method	Objective	Runtime (s)
GAR LNS	3.669731750690e+10	11.72607421875
Gurobi	1.973028493446e+10	64.72920036315918
TRPO (300 timesteps)	1.497809024771e+10	46.93
PPO (500 timesteps)	3.076277074907e+09	216.86
PPO (1000 timesteps)	3.472219655817e+08	993.22

It can clearly be seen that there is a significant improvement in runtime for Graph Augmented Random LNS when compared to Gurobi, 11.73 seconds to 64.73 seconds = an improvement of 82%. Similarly, there is a significant improvement in runtime for Gurobi when compared to PPO(500), 216.86 seconds to 64.73 seconds = an improvement of 70%. Again, there is an improvement in runtime for PPO(500) when compared to PPO(1000), 993.22 seconds to 216.86 seconds = an improvement of 78%. For TRPO to produce similar objective as GAR LNS and Gurobi, we require around 45 seconds runtime,



(a) Average GAR LNS Path (b) Average Gurobi Path

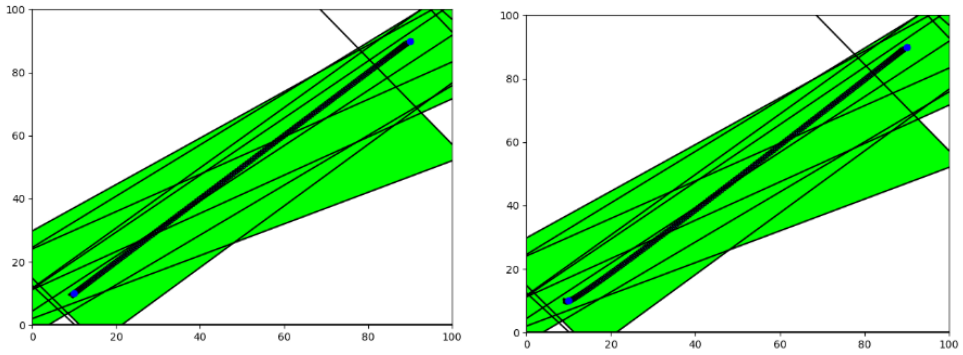
Figure 4: Path comparison for Graph Augmented Random LNS and optimal Gurobi solution on an average environment

which is a 30% improvement over Gurobi. However, GAR LNS has a 75% improvement over TRPO. The objective value was worse by a factor of 1.86 between GAR LNS and Gurobi, by a factor of 6.4 between Gurobi and PPO(500), and by a whopping factor of 8.9 between PPO(500) and PPO(1000). Again, the objective produced by PPO(500) that of TRPO(500) and the same goes for PPO(1000) and TRPO(1000). We can perhaps generalize that in an average environment, runtime and objective values are negatively correlated.
From what can be seen in the plotted paths, the trajectories for Graph Augmented Random LNS and Gurobi are very similar.

For simpler environments, Graph Augmented Random LNS still achieved a high runtime improvement, with less objective loss:

Table 2: Objective and Runtime Comparison on Simple Environment

Method	Objective	Runtime (s)
GAR LNS	1.964072827949e+10	1.0885906219482422
Gurobi	1.847049320428e+10	5.37230110168457



(a) Simple GAR LNS Path (b) Simple Gurobi Path

Figure 5: Path comparison for Graph Augmented Random LNS and optimal Gurobi solution on a simple environment

From these values we see that GAR LNS still maintained an 80 percent speedup on runtime, with minimal objective loss (less than 7 percent), with almost identical paths. Of course, the design of this environment made solving the problem almost trivial, but we still found some improvement.

For some more difficult environments where the optimal solution took over one minute to find, Graph Augmented Random LNS sometimes failed to find a solution within the given time limit, likely due to the fact that going from one convex region to another on that path had a difficult solution.

4.2 REINFORCE (Monte-Carlo policy gradient)

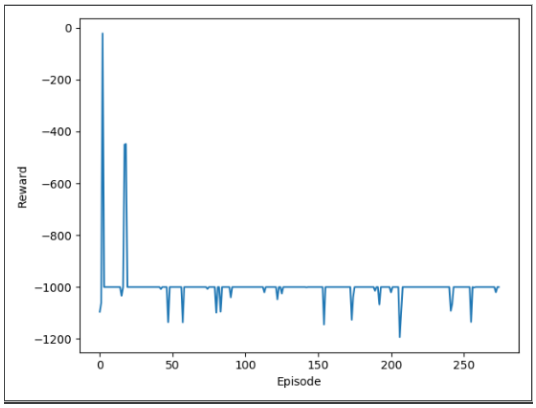


Figure 6: Reinforcement Learning Results

In fig. 6 you can find the performance of our neural network model after 250 iterations. Notice that the reward flat lines at around -1000 which is the minimum amount of reward one step of the RL environment will produce. This happens in the case where the policy produced an infeasible assignment.

5 Conclusion and Discussion

From testing our implementation of Graph Augmented Randomized LNS, we were able to see a significant speedup in runtime, with losses in the objective that increased with the complexity of the environment. This effect was compounded in the performance of the TRPO and PPO algorithm. When plotting the generated paths, the impact of this seemed to be marginal when looking at the differences in the real world paths. This shows that utilizing LNS and related methods to speedup robot MIP planning does have a significant speedup (around 80 percent, persisting across different complexities of environments), and while mathematically the objective loss increases, in the real world this may not matter as much. Overall, PPO, TRPO, and Graph Augmented Randomized LNS seems to be a good approach for solving MIPs for robot locomotion, for problems up to a certain complexity. Environments that are too complex will either have a loss in objective that outweighs the speedup in runtime, or no run-time speed up at all due to the algorithm being unable to solve the problem. This problem becomes especially acute even in environments with solutions but that cause numerical issues due to solutions that lie near a constraint.

With the REINFORCE algorithm, we did not find positive results. After close inspection it appears our policy overfit the reasonable rewards in the few episodes so that it would always choose the same regions even if they weren't feasible. As future work we would like to experiment with reducing the complexity of the model and perhaps adding other layers that reduce over-fitting such as dropout layers.

Ultimately, we think that, despite the negative result with reinforcement learning, using LNS to speed up MIPs in the context of trajectory optimization can bring substantial benefits. With minimal tuning, a random selection policy outperformed Gurobi by 80%. It is not unreasonable to think that a better approach could bring MIP trajectory optimization problems into the realm of real-time robotics. With this we have presented a foundation on how to use Large Neighborhood Search and Reinforcement Learning in the context of trajectory optimization.

References

[1] Kai Arulkumaran, Antoine Cully, and Julian Togelius. "Alphastar: An evolutionary computation perspective". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2019, pp. 314–315.

[2] Robin Deits and Russ Tedrake. "Efficient mixed-integer planning for UAVs in cluttered environments". In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 42–49.

[3] Robin Deits and Russ Tedrake. "Footstep planning on uneven terrain with mixed-integer convex optimization". In: 2015 (Feb. 2015), pp. 279–286. DOI: 10.1109/HUMANOIDS.2014.7041373.

[4] He He, Hal Daume III, and Jason M Eisner. "Learning to search in branch and bound algorithms". In: *Advances in neural information processing systems*. 2014, pp. 3293–3301.

[5] Elias Khalil et al. "Learning Combinatorial Optimization Algorithms over Graphs". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 6348–6358. URL: http://papers.nips.cc/paper/7214-learning-combinatorial-optimization-algorithms-over-graphs.pdf.

[6] Sergey Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), pp. 1334–1373. ISSN: 1532-4435.

[7] Xue Bin Peng et al. "DeepMimic". In: *ACM Transactions on Graphics* 37.4 (Aug. 2018), pp. 1–14. ISSN: 1557-7368. DOI: 10.1145/3197517.3201311. URL: http://dx.doi.org/10.1145/3197517.3201311.

[8] *Re-create iris+quadrotor mi-convex planning examples · Issue #6243 · RobotLocomotion/drake*. GitHub. Library Catalog: github.com. URL: https://github.com/RobotLocomotion/drake/issues/6243 (visited on 06/06/2020).

[9] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[10] John Schulman et al. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[11] David Silver et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359.

[12] Jialin Song et al. *A General Large Neighborhood Search Framework for Solving Integer Programs*. 2020. arXiv: 2004.00422 [math.OC].

[13] Russ Tedrake and the Drake Development Team. *Drake: Model-based design and verification for robotics*. 2019. URL: https://drake.mit.edu.