Learning Proposal Distributions for Refining Symbolic Plans

Rohan Chitnis[1], Dylan Hadfield-Menell[2], and Pieter Abbeel[2]

***Abstract*— Long-horizon tasks in robotics require an agent to reason through a long sequence of high level steps in order to reach a goal, as well as determine appropriate joint trajectories for executing each step. Planning for and executing these tasks is a key challenge in the field. Recent work has contributed to hierarchical planning, which seeks to plan for such goals by combining task and motion planning. These planning systems rely on methods to propose candidate values for instantiating continuous variables in a symbolic plan, a process known as *plan refinement*. A core limitation of task and motion planning systems is the manner in which these values are sampled: using hand-coded distributions that leverage specificity about the geometry of the environment and its objects. In this paper, we apply techniques in reinforcement learning to avoid hand-coding these distributions, instead learning ones that tend to propose good values. Our contributions are: 1) a novel randomized local search algorithm for plan refinement; 2) a formulation of plan refinement in the reinforcement learning framework; and 3) experiments showing that our system, using learned sampling distributions, achieves comparable to improved performance versus hand-coded distributions in a variety of test environments.**

I. Introduction

A long-term goal of robotics research is the introduction of intelligent household robots. Such robots would face complex tasks, such as setting dinner tables and doing laundry. Planning for these long-horizon tasks is infeasible using only motion planning due to the sheer number of steps involved, making apparent the need for a hierarchical system of reasoning.

Recent methods for hierarchical planning focus on the intersection of high level *task* planning and low level *motion* planning [1], [2], [3]. In this framework, the (classical) task planner produces a symbolic plan containing a sequence of actions to reach a goal state, and heuristic sampling techniques propose concrete values for the continuous variables in the plan, thus grounding it. This process of assigning candidate values to the plan variables is known as *plan refinement*. These candidate values are then checked locally for feasibility by calling a motion planner.

This hierarchy enforces abstraction between the role of the task planner and that of the motion planner: the task planner maintains no knowledge of the environment geometry, and the motion planner has no understanding of the overall goal. A central challenge in building such a system is designing good heuristics for the sampling in plan refinement.

Our work improves directly upon the task and motion planning system presented by Srivastava et al. [1]. They propose an *interface layer* for plan refinement, which performs an exhaustive backtracking search over a discrete set of parameter values for refining the symbolic plan into a set of collision-free trajectories. If a motion planning feasible refinement is not found within the resource limit, symbolic error information is propagated back to the task planner, and a new symbolic plan is produced. The system uses an off-the-shelf classical task planner and motion planner, both as black boxes. We build upon this framework because it is easy to separate the plan refinement module from the task planning module.

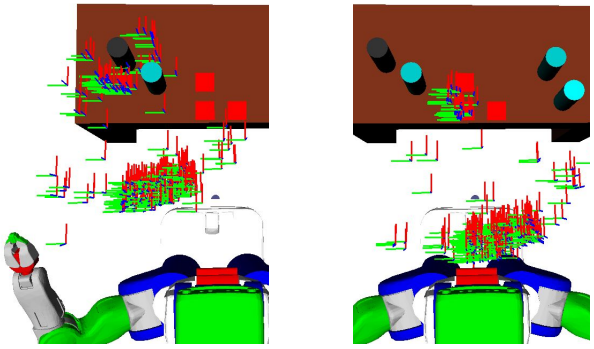


Fig. 1: Screenshots showing some distributions learned by our system in one experimental setup. The robot is tasked with grasping the black cylinder and putting it down on the central red square. The left image shows learned base motion and grasping distributions, and the right shows learned base motion and putdown distributions. The grasp distribution properly learned to avoid the region close to the blue obstruction. We sample from these distributions to refine the high level plan, rather than relying on hand-coded ones.

A key limitation of task and motion planning systems is that they typically rely on hand-coded heuristic sampling distributions for plan refinement. These distributions are tuned for the specific geometric properties of the environment and its objects. This restriction has several negative implications: the parameters of the hand-coded distributions must be fine-tuned when running the system in a new setting, and the resulting refinement distributions lack robustness.

In this work, we apply reinforcement learning to find good proposal distributions for symbolic plan refinement. We take inspiration from the work of Zhang and Dietterich [4], who applied reinforcement learning to planning problems in a job shop scheduling setting. Using methods adapted from Zucker et al. [5], who train a configuration space sampler for motion planning using features of the discretized workspace,

[1] ronuchit@berkeley.edu
[2] {dhm, pabbeel}@eecs.berkeley.edu

we train a policy that determines how to sample values for plan symbols, using policy optimization with linear function approximation. Our approach allows the learned distributions to be continuous, robust to changes in the environment, and trainable for any experimental setting, eliminating the need for hand-tuned distribution parameters.

The three contributions of our work are as follows:

1) We present randomized refinement, a novel local search algorithm for plan refinement. Randomized refinement maintains at all times a set of values for all symbolic plan variables, then at each iteration randomly resamples one whose current instantiation is causing a failure. This naturally guides plan variable resampling intelligently and easily lends itself to reinforcement learning.
2) We describe how to formulate plan refinement (using randomized refinement) in the standard reinforcement learning framework, so that sampling distributions for plan variables can be learned instead of hand-coded. Our formulation optimizes for minimizing the number of calls to the motion planner.
3) We present experiments to evaluate our approach in a variety of test environments involving complicated grasp and putdown actions with obstructions and base motion. Our experimental results demonstrate comparable to improved performance for both motion planning time and number of calls to the motion planner, when compared to the hand-coded distributions used in the original system.

II. Related Work

Prior work has been devoted to using machine learning techniques for training heuristics to guide search algorithms. This general formulation has been applied to many domains.

Boyan et al. [6] present an application to solving optimization problems using local search routines. They describe an algorithm, STAGE, for learning a state evaluation function using features of the optimization problem. This function then guides the local search toward better optima.

Arbelaez et al. [7] use machine learning to solve constraint satisfaction problems (CSPs). Their approach, Continuous Search, maintains a heuristic model for solving CSPs and alternates between two modes. In the *exploitation* mode, the current heuristic model is used to solve user-inputted CSPs. In the *exploration* mode, this model is refined using supervised learning with a linear SVM.

Xu et al. [8] apply machine learning for classical task planning. They draw inspiration from recent advances in discriminative learning for structured output classification. Their system trains heuristics for controlling forward state-space beam search in task planners.

To our knowledge, our work is the first to use reinforcement learning for plan refinement in a task and motion planning system.

III. Background

We provide relevant technical background and introduce notation used throughout the paper.

A. Reinforcement Learning

A reinforcement learning problem is typically formulated as a Markov decision process (MDP). An MDP is defined by the following:

- A state space S.
- An action space A.
- A transition function $P(s'|s,a)$ for $s, s' \in S, a \in A$.
- A reward function $R(s,a,s')$ for $s, s' \in S, a \in A$.
- A discount factor $\gamma \in [0,1]$.

The goal of a reinforcement learning problem is to train a policy $\pi(s) : S \rightarrow A$ that selects which action $a \in A$ to execute given a current state $s \in S$, in order to maximize the sum of expected discounted future rewards.

B. Reinforcement Learning for Planning

Zhang and Dietterich [4] present a technique for formulating the job shop scheduling problem using the reinforcement learning framework. The goal of job shop scheduling is to find a minimum-duration schedule of a set of jobs with temporal and resource constraints. They define the following:

- S is all temporal configurations and resource allocations of the jobs.
- A is changing either the time or resource allocation of a single job.
- R is a measure of schedule goodness based on its length relative to the difficulty of the scheduling problem.

They then use the temporal difference learning algorithm $TD(\lambda)$ [9] to learn a heuristic state evaluation function, used in a one-step lookahead search to produce good schedules. Their approach outperforms the previous state of the art for this task; it also scales better to larger scheduling problems after having been trained only on smaller ones. This application of reinforcement learning to planning has been built on extensively in the literature, applied to domains such as retailer inventory management [10], single-agent manufacturing systems [11], and multi-robot task allocation [12].

Zucker et al. [5] use reinforcement learning to find good sampling distributions for motion planning. Their approach uses features of a discretization of the workspace to train a non-uniform configuration space sampler. They use stochastic policy gradient updates to learn a good weight vector θ for these features. Toward this end, they define the distribution

$$q(\theta, x) \propto exp(\theta^T f(x)) \tag{1}$$

They also define the expected reward under q of an episode ξ with length ϵ:

$$\eta(\theta) = E_q[R(\xi)] \tag{2}$$

and provide an approximation for its gradient:

$$\nabla\eta(\theta) \approx \frac{R(\xi)}{\epsilon} \sum_{i=1}^{\epsilon} (f(x_i) - E_q[f]) \tag{3}$$

where $E_q[f]$ is the feature vector expectation under q. The weight vector update is then:

$$\theta \leftarrow \theta + \alpha\nabla\eta(\theta) \tag{4}$$

for appropriate step size α.

C. Task and Motion Planning

We formulate a task and motion planning problem as follows. A fully observed task planning problem is a tuple $(\mathcal{O}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{C})$:

$\mathcal{O}$ is a set of symbolic references to *objects* in the environment, such as cylinders, cylinder locations, and grasping poses. These are treated as strings.

$\mathcal{F}$ is a set of fluents, which define relationships among objects and hold either true or false in the state.

$\mathcal{I}$ is the set of fluents that hold true in the initial state. All others are assumed false.

$\mathcal{G}$ is the set of fluents defining the goal state.

$\mathcal{C}$ is a set of parametrized *high level actions*, defined by *preconditions*, a set of fluents characterizing what must hold true in the current state for the action to be validly performed, and *effects*, a set of fluents that hold true after the action is performed.

A solution to a task planning problem is a valid sequence of actions $a_0, a_1, ..., a_n \in \mathcal{C}$ which, when applied to the state of fluents successively starting with $\mathcal{I}$, results in a state in which all fluents of $\mathcal{G}$ hold true. We refer to this sequence of actions as a *symbolic*, or *high level*, plan: the task planner maintains no knowledge about the geometric meaning behind any of the symbols.

Any continuous variables of a high level plan, such as grasping poses for objects, are abstracted into symbols representing their nature. Task planners are, thus, unable to maintain a geometric interpretation for these symbols, so we use hierarchical task and motion planning to provide this interpretation. We adopt the approach of Srivastava et al. [1]. They describe an *interface layer* responsible for assigning continuous values to the high level plan variables, discovering geometric facts about the environment, and sending these facts back to the task planner to be added into the fluent state. This process of assigning such continuous values to the abstract plan variables is known as *plan refinement*. The authors introduce an algorithm for plan refinement called TRYREFINE, an exhaustive backtracking routine over feasible high level plan parameter values.

IV. RANDOMIZED REFINEMENT

In order to move toward refining high level plans using continuous sampling distributions, we present one contribution of our work: randomized refinement, a local search algorithm for plan refinement. It maintains at all times a value for each high level plan variable; at each iteration, a variable whose current value is leading to a motion planning failure or action precondition violation is picked randomly and resampled locally. This method of making local improvements draws parallels with the job shop scheduling algorithm and is straightforward to formulate in the reinforcement learning framework, as we show in the next section. Algorithm 1 shows pseudocode for randomized refinement.

The procedure takes two arguments, the high level plan object and a maximum iteration count. In Line 2, all variables in the high level plan are initialized by sampling from the corresponding distributions. For efficiency, we only initialize the symbolic pose references (such as object grasping poses) with IK-feasible values, so as to avoid unnecessary calls to the motion planner. We then call the MOTIONPLAN subroutine in Line 4, which iterates through the sequence of actions that comprise the high level plan. For each action, a call to the motion planner is made (Line 22) using the instantiated values for the parameters of that action, to attempt to find a trajectory linking the sampled poses. If this succeeds, the preconditions of the action are tested (Line 26). As part of this step, we verify that the trajectory returned by the black box motion planner is collision-free, satisfying the precondition that the trajectory is feasible in the environment.

Thus, based on the returned values of MOTIONPLAN, we may distinguish a motion planning failure from a precondition violation, for the current refinement. We appropriately call the `resample` routine on the high level parameters associated with the failure; this routine picks one of the parameters at random and resamples it from its refinement

Algorithm 1 Randomized refinement.

```
1:  procedure RANDREF(HLP, N)
2:      init ← initRefinement(HLP)
3:      for iter = 0, 1, ..., N do
4:          failStep, failPred ←MOTIONPLAN(HLP)
5:          if failStep is null then
6:              return success
7:          else
8:              if failPred is null then
9:                  # Motion planning failure.
10:                 failAction ← HLP.ops[failStep]
11:                 resample(failAction.params)
12:             else
13:                 # Action precondition violation.
14:                 resample(failPred.params)
15:             end if
16:         end if
17:     end for
18:     Raise failure to task planner, receive new plan.
19: end procedure
20: procedure MOTIONPLAN(HLP)
21:     for op_i in HLP.ops do
22:         res ← op_i.motionPlan()
23:         if res.failed then
24:             return i, null
25:         else
26:             failPred ← op_i.checkPreconds()
27:             if failPred then
28:                 return i, failPred
29:             else
30:                 # Found successful plan refinement.
31:                 return null, null
32:             end if
33:         end if
34:     end for
35: end procedure
```

distribution. Again for efficiency, we keep resampling until we have an IK-feasible value. If we reach the iteration limit (Line 18), we convert the most recent failure information into a symbolic representation, then raise it to the classical planner, which will update its fluent state and provide a new high level plan.

We emphasize the benefits of randomized refinement. First, it lends itself to easy formulation as an MDP, as we show in the next section. Additionally, it allows the parameter instantiations for a particular action in the plan to be influenced by those for a *future* action. For example, in a pick-and-place task, it can make sense for the object's grasp pose to be sampled conditionally on the current instantiation of the putdown pose, even though the putdown appears after the grasp in the plan sequence. Thus, it is easy for plan refinement to respond to long-term dependencies in continuous values of plan variables. Finally, its simplicity allows for ease of debugging.

V. Learning Refinement Distributions

In this section, we present our primary contribution: a method for training good sampling distributions using reinforcement learning. We first describe how to formulate plan refinement in the reinforcement learning framework. Then, we describe the training process and how to learn a policy for plan refinement.

A. Formulation as Reinforcement Learning

We formulate plan refinement as an MDP as follows:

- A state $s \in S$ is a tuple (P, r_{cur}, E), consisting of the symbolic plan, its current refinement (instantiation of values for all plan parameters), and the environment.
- An action $a \in A$ is a pair (p, x), where p is the discrete-space plan parameter to resample and x is the continuous-space value to assign to p. We note that randomized refinement defines a fixed policy for selecting p, using motion planning failure and precondition violation information to guide its decision. Thus, we focus on training a policy for choosing x, by defining a sampling distribution (based on the state) for each plan parameter.
- Under these definitions, the transition function is deterministic.
- Our reward function $R(a, s')$ provides constant rewards based on parameter resampling. Because we check for IK feasibility when resampling, we provide -1 reward every time we sample an IK-infeasible pose, and $+3$ reward for an IK-feasible pose. Similarly, we give -3 reward upon motion planning failure or action precondition violation, and $+5$ reward upon a successful call to the motion planner for an action. We do not provide any bonus reward for being in a state where the entire plan refinement is valid. This choice of R implies that our training optimizes the sampling distributions for motion planning feasibility, rather than explicitly for refining plans. However, the two are closely correlated – randomized refinement succeeds in fewer iterations when the sampling distributions produce more motion planning feasible values.

B. Training Process

We now show how to apply the method of Zucker et al. [5] to learn a policy for plan refinement. We train a weight vector θ_p, associated with the plan parameter p, for each parameter *type* (e.g. "left gripper grasp pose" or "right gripper putdown pose"), so that the learned distributions are not tied to any single high level plan. These weight vectors encode the policy for selecting x for actions $a \in A$, as described in the previous subsection. Our feature function $f(s, p, x)$ maps the current state $s \in S$, plan parameter p, and sampled value x for p to a feature vector; f defines a policy class for our problem.

Our training system is defined by the following:

$\mathcal{P}$: A distribution over planning problems, encompassing both the task planning problem and the environment.
θ_p: Described above.
R: The reward function, described in the previous subsection.
$f(s, p, x)$: Described above.
N: The number of planning problems on which to train.
L: The number of samples for one planning problem.
ϵ: The number of samples comprising an episode. We perform a weight vector update after each episode.

The training is a natural extension of randomized refinement and progresses as follows. N times, sample from $\mathcal{P}$ to obtain a complete planning problem Π. For each Π, run the randomized refinement algorithm to attempt to find a valid plan refinement, allowing the `resample` routine to be called L times. As the system takes on-policy actions $a \in A$ (based on current θ values) for variable resampling, collected rewards are appended to a global reward list. If a valid, motion planning feasible refinement is found, we forcibly resample one of the high level variables at random (potentially making the refinement infeasible again) and continue. Every ϵ calls to `resample`, we perform a gradient update on the weight vectors θ using the collected rewards and samples since the previous update.

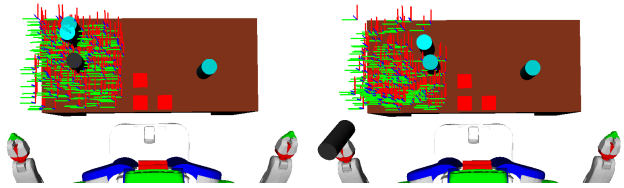
This process evinces the strength of randomized refinement for our system: only the variable instantiations responsible for failures are resampled, so the learning is naturally focused toward improving sampling distributions which are not yet well-shaped.

We adopt the distribution in (1) for a parameter p with sample value x, in state $s \in S$:

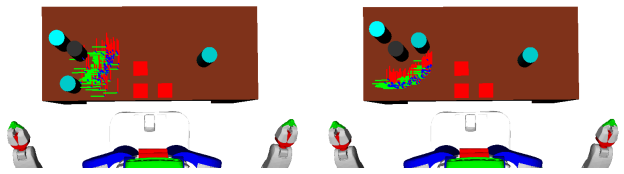
$$q(s, p, x) \propto exp(\theta_p^T f(s, p, x)) \tag{5}$$

and update the θ_p according to (2) - (4).

We sample x from q using the Metropolis algorithm, an MCMC technique for sampling from a probability distribution using a symmetric proposal distribution. At each iteration, the sampled value is based only on the previous one, so the samples form a Markov chain. Also, since our distributions are continuous, we cannot easily calculate

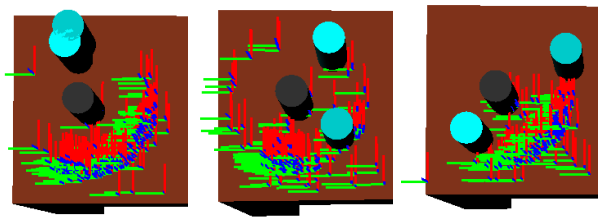


(a) Initial distribution (b) After 5 iterations.



(c) After 15 iterations. (d) Final distribution.

Fig. 2: Learned left arm grasping distributions used to pick up the black can, after different training iterations. An iteration refers to a single run of randomized refinement, which terminates after L calls to the resample function. The initial distribution is uniform because we initialize weights to $\vec{0}$. The final distribution is after 20 iterations.



(a) (b) (c)

Fig. 3: The final distribution from Figure 2 is shown with different obstruction layouts. The system learns to avoid obstructions while providing a good set of grasping poses.

$E_q[f]$, so we approximate it by averaging together the feature vectors for several samples from q.

The distribution q represents the policy we are training: it describes how to sample a continuous-space value x for a parameter p, given current state s. Since we learn weights for a linear combination of the features, we have employed policy gradient techniques with linear function approximation.

VI. Experiments

A. Evaluation

We evaluate our reinforcement learning system in several pick-and-place tasks, varying the number of obstructions, obstruction locations, and whether the base is allowed to move. We compare performance with that of the hand-coded sampling distributions used in Srivastava et al. [1]. Because our learned distributions are optimized for producing motion planning feasible samples, we do not evaluate performance on total system running time; rather, we evaluate the refinement of only the final high level plan. This is because, if we have a high level plan with *no* valid refinement, our system may find more IK-feasible samples (and thus trigger more calls to the motion planner) than the baseline system. This causes increased time spent on refinement attempts before raising error information back to the task planner, which should not be penalized. The error propagation occurs when the iteration limit is reached in the randomized refinement algorithm.

System	Scenario	% Solved	Avg MP Time (s)	Avg # MP Calls
B	1	100	8.16	18.93
L	1	96	6.11	11.5
B	2	96	8.74	19.7
L	2	94	12.26	20.77
B	3	84.44	10.77	23.16
L	3	97.78	10.6	19.1
B	4	33.33	12.1	21.1
L	4	96.67	9.19	17.78
B	5	100	4.52	12.24
L	5	100	5.94	15.36

TABLE I: Percent solved, along with time spent motion planning and number of calls to the motion planner for the final refinement. Results are averaged across 50 test environments per scenario, when both the baseline and our system succeed. The test environments were grouped into batches of 5, and each batch first received independent training with a distinct random seed. For each scenario, we provide baseline system results using hand-tuned sampling distributions (B) and results from running our reinforcement learning system (L).

We use the reward function described earlier. Our weight vectors are initialized to $\vec{0}$ for all pose parameter types: grasp pose for each arm, putdown pose for each arm, and base motion. The initialization represents a uniform distribution across the limits of the geometric search space. For grasp and putdown actions, the limits are a cube of side length 30 centimeters around the target object or putdown location. For base motion actions, the limits are a square of side length 1 meter around the object or location which the robot is approaching. Currently, our feature function f incorporates only geometric aspects of the sample and environment, such as distance between the sample and object being grasped, and distance from the sample to nearby obstructions.

Our experiments are conducted in Python 2.7 using the OpenRave simulator [13] with a PR2 robot. The motion planner we use is trajopt [14], and the task planner is Fast-Forward [15]. The experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM.

We train and test five different scenarios, all of which share a common goal of grasping a target object from a table and putting it down at a specific location on the table:

Scenario 1: Target object is surrounded by 1 obstruction.
Scenario 2: Target object is surrounded by 2 obstructions.
Scenario 3: Target object is surrounded by 3 obstructions.

Scenario 4: Target object is surrounded by 1 obstruction, and putdown location is obstructed in cardinal directions.

Scenario 5: Target object is surrounded by 1 obstruction, but the robot begins out of reach of the table and must learn base motion.

For the first 4 scenarios, we use $N = 20$, $L = 16$, and $\epsilon = 4$. For the final one involving base motion, we use $N = 60$, $L = 100$, and $\epsilon = 20$, allowing randomized refinement to run for more iterations because the high level plan is longer. The locations of the obstructions surrounding the target object were sampled uniformly from a thick ring around the object with inner radius 0.13 meters and outer radius 0.25 meters. Table I summarizes our quantitative results. Figure 1 plots learned distributions for a move-grasp-move-putdown sequence of actions. Figure 2 illustrates the training of a grasp distribution, and Figure 3 shows how the learned distribution interacts with obstructions.

B. Discussion

The quantitative results indicate that our system performs comparably to the baseline in all five environment setups. Scenario 4, for which our learning algorithm greatly outperforms, was designed to be difficult for the baseline system, because the putdown pose sample space is discretized such that the gripper always approaches the putdown location from a cardinal direction. These cardinal directions were blocked by obstructions, meaning the baseline system will generally perform poorly in this environment, illuminating its lack of robustness.

VII. Conclusion and Future Work

We presented a novel application of reinforcement learning to plan refinement in task and motion planning. Our method trained a policy for refining symbolic plans by learning good sampling distributions for plan parameters. The choice of which parameter to resample at each iteration was governed by a novel refinement strategy we presented called randomized refinement. We evaluated performance by comparing against a baseline of hand-coded distributions for several challenging pick-and-place tasks. Our system demonstrated overall comparable to improved performance in motion planning time and number of motion planner calls.

One important direction for future work is improving the feature vector to incorporate information about the symbolic plan and previous samples of a parameter. Another is training a system that decides whether to return to the high level on its own, instead of relying on reaching the iteration limit in the randomized refinement algorithm. This would make the training optimize more directly for producing a valid refinement, given an arbitrary task planning problem.

References

[1] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," *IEEE Conference on Robotics and Automation*, 2014.

[2] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1470–1477.

[3] F. Lagriffoul, D. Dimitrov, J. Bidot, A. Saffiotti, and L. Karlsson, "Efficiently combining task and motion planning using geometric constraints," 2014.

[4] W. Zhang and T. G. Dietterich, "A reinforcement learning approach to job-shop scheduling," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1114–1120. [Online]. Available: http://dl.acm.org/citation.cfm?id=1643031.1643044

[5] M. Zucker, J. Kuffner, and J. A. D. Bagnell, "Adaptive workspace biasing for sampling based planners," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, May 2008.

[6] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *J. Mach. Learn. Res.*, vol. 1, pp. 77–112, Sep. 2001. [Online]. Available: http://dx.doi.org/10.1162/15324430152733124

[7] A. Arbelaez, Y. Hamadi, and M. Sebag, "Continuous search in constraint programming," in *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion, Eds. Springer Berlin Heidelberg, 2012, pp. 219–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21434-9_9

[8] Y. Xu, S. Yoon, and A. Fern, "Discriminative learning of beam-search heuristics for planning," in *PROCEEDINGS OF THE INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE 2007*, 2007, pp. 2041–2046.

[9] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[10] B. Van Roy, D. P. Bertsekas, Y. Lee, and J. N. Tsitsiklis, "A neuro-dynamic programming approach to retailer inventory management," in *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, vol. 4. IEEE, 1997, pp. 4052–4057.

[11] Y.-C. Wang and J. M. Usher, "Application of reinforcement learning for agent-based production scheduling," *Engineering Applications of Artificial Intelligence*, vol. 18, no. 1, pp. 73 – 82, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197604001034

[12] T. S. Dahl, M. Matarić, and G. S. Sukhatme, "Multi-robot task allocation through vacancy chain scheduling," *Robotics and Autonomous Systems*, vol. 57, no. 6, pp. 674–687, 2009.

[13] R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34, July 2008.

[14] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization." in *Robotics: Science and Systems*, vol. 9, no. 1. Citeseer, 2013, pp. 1–10.

[15] Jörg Hoffman, "FF: The fast-forward planning system," *AI Magazine*, vol. 22, pp. 57–62, 2001.