Generalizing the knowledge from the past for informed motion planning

Anonymous. Paper-ID [add your ID here]

I. Introduction

The ability to intelligently plan a course of action in an unstructured and unforeseen environment is crucial in creating an intelligent robot. A planner seeks to achieve this goal: it takes as inputs the environment dynamics, initial and goal states, and a score function to produce a plan, which consists of trajectory of robot actions that achieve an objective of interest. An exemplar planner would generate a high quality plan, measured by the given score function, as quickly as possible.

While we have made great progress in planning algorithms themselves, such as motion planners, task planners, or trajectory optimizers [cite some papers here], relatively little attention has been made in generalizing the knowledge gained from solving past planning problem instances to a new instance. Figure 1 illustrates this point with an example. Here, the robot needs to transport the black object to the table on the other side while going through the narrow passage where it requires the robot to turn side ways. If we can learn that we had to turn sideway from past planning problem instance, then it would save a great amount of computation time for a new problem instance.

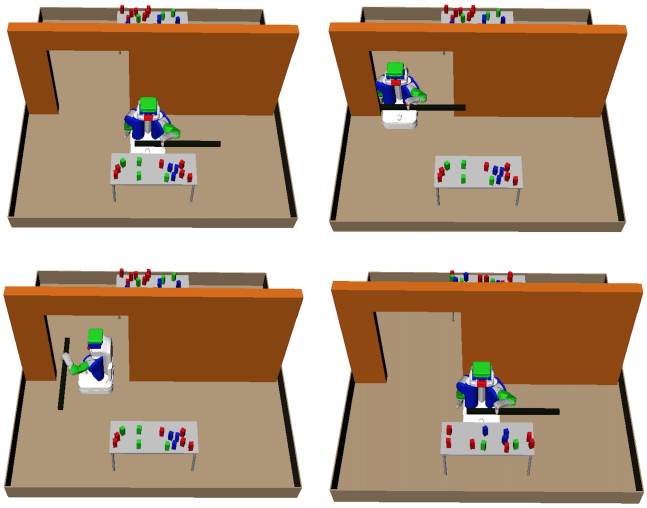

Fig. 1: The top two figures in illustrate that this environment requires you to turn side ways in order to get through the passage, as shown in bottom left. Bottom right is a new problem instance, with different arrangement of obstacles on the two tables. Can we generalize the knowledge that we need to turn side ways from the past problem instance to this one?

Current planning algorithms do no try to exploit such past knowledge and always plan from scratch. They spend time on re-doing what it has possibly done before, which could be used to find a better quality plan. Hence it would be extremely desirable to be able to adapt such knowledge to a new planning problem instance intelligently, as to improve the speed and quality of planning.

In light of this, we introduce the notion of *solution contraint* as a representation of such past knowledge that informs the planner to more quickly generate high quality plans. A solution constraint could be a subgoal to quickly plan a motion through a narrow passage, a grasp to robustly picking an object, or the closest feasible object pose for placing it on a region of interest. We believe this is more compact and generalizable representation of past planning knowledge than the plans themselves, which does not generalize well to complex problems that we are considering. Hereafter, we will refer to a solution constraint as constraint whenever trivial from the context.

The main challenge in selecting the best solution constraint for the current problem instance based on the past planning solutions is that there is no notion of metric in the space of planning problem instances. Instead of constructing such representation, we directly characterize planning problem instances with a library of solution constraints and their scores in each of the problems. The main intuition here is in viewing the surrounding environment, which defines a planning problem instance, as a black box function which outputs a numerical evaluation of the robot's plan associated with the constraint, in lieu of representing it with some sensory data, which can be misleading. The relationship between problem instances then can be constructed based on these values.

More specifically, using this library of solution constraints, we make the the assumption that the score functions for the problem instances are sampled from the same distribution. This assumption allows us to construct an upper confidence bound (UCB) for the scores of the new problem instance, using statistics of the past values. Based on this bound, we propose BOX (Blackbox Optimization with Experience), an UCB-type algorithm that tries to find a good solution constraint for the new scene.

We evaluate the performance BOX in three different domains. In the first domain, the robot needs to pick an object up by choosing a grasp from a discrete set. Its objective is to choose the best grasp such that when the motion planner is constraint to plan a path to the selected pregrasp pose, the path maximizes the distance from the obstacles. A planning problem instance is defined by the random arrangement of

obstacles. In the second domain, the robot is faced with the same task, but in addition to grasp it needs to choose a base pose from a continuous space. A problem instance is again defined by the arrangement of obstacles. In the last domain, the robot has to transport an object of random length from one table to another, where there is a narrow passage in between. A solution constraint for this domain consists of a grasp for picking up the object, an object placement pose, robot base pose for placing the object, and subgoals for motions to pregrasp pose, to the target base pose, and placement pose. A problem instance is defined by both the length and arrangement of obstacles. The objective for this domain is to minimize the sum of the lengths of three paths. We compare BOX to four other algorithms, and show that BOX outperforms them both in terms of time and quality of the plans that it produces in all domains.

II. Algorithm

The problem that we are trying to solve is a black box function optimization of solution constraints, with knowledge of values of the past functions. More specifically, we are facing a planning problem z, and we would like to optimize the function it induces, with the knowledge of the values of past functions induced by $z_1, \cdots, z_n$. The main assumption here is that all the problem instances are sampled from the same probability distribution, i.e. $z \sim P(z)$.

Assume, for now, that we have been given a "training set" in the form of n problem instances $z_1, \ldots z_n$ drawn from $P(z)$ and m solution constraints $\Theta = \{\theta_1, \ldots, \theta_m\}$, chosen possibly because they were good or optimal for problem instances in the training set.

$$\mathfrak{J} = \begin{bmatrix} J^{z_1}(\theta_1) & \cdots & J^{z_1}(\theta_m) \\ J^{z_2}(\theta_1) & \cdots & J^{z_2}(\theta_m) \\ \vdots & \vdots & \vdots \\ J^{z_n}(\theta_1) & \cdots & J^{z_n}(\theta_m) \end{bmatrix}$$

Here, $J^{z_i}(\theta_j)$ is the score of executing a plan with θ_j in a training planning problem instance z_i. That is, given θ, $J^{z_i}(\theta)$ is a wrapper around a motion planner that calls the planner which returns a plan P_θ associated with the constraint θ, and then evaluates this plan using the underlying score function $J^{z_i}(P_\theta)$.

Now, given a new problem instance, z, our goal is to efficiently find a high-scoring constraint from Θ without evaluating all of them. We will, particularly, explore a class of procedures in which we evaluate $J^z(\theta)$ for $k << m$ values of θ in Θ. It does so by exploiting the patterns among $J^z(\theta)$ values, both across different z for a fixed θ and across different θ values for a fixed z.

Our strategy will be to construct upper confidence bounds on the score of each of the unevaluated soultion constraint θ_i on the new problem instance z, and evaluate the one with the highest upper confidence bound. We will do so for k rounds and ultimately return the highest-scoring plan P_{θ^*} that was constructed during the process of evaluating the k selected constraint. This strategy is effective in many problems that involve optimization under uncertainty, selecting the next candidate for evaluation that has the most "potential" to be the best.

First, we make use of the intuition that some constraints (via the plans they generate) are better than others, in expectation over problem instances drawn from $P(z)$. So, if we knew some statistics of that distribution it would be possible principle to use Chebyshev's inequality to construct a bound of the form

$$J^z(\theta_i) \leq E_z[J^z(\theta_i)] + C_{1,i}\sigma_z[J^z(\theta_i)]$$

which holds with probability $(1/C_{1,i})^2$, for each θ_i. Then, we define problem-wise upper bound as

$$B_p(\theta_i, z) = E_z[J^z(\theta_i)] + C_{1,i}\sigma_z[J^z(\theta_i)] \tag{1}$$

The interpretation of this upper bound is simple: we should try the solution constraint that has a high expected value and variance first, because if a constraint has low variance and low expected value, then this indicates it consistently performs worse than other constraints.

Second, we exploit correlation between the scores of different constraints on the same problem instance. We again use Chebyshev's inequality to construct a bound on the difference between the scores of constraints i and j on the problem instance z:

$$J^z(\theta_i) \leq J^z(\theta_j) + \mathbb{E}_z[J^z(\theta_i) - J^z(\theta_j)] + C_{2,i} \cdot \sigma_z[J^z(\theta_i) - J^z(\theta_j)]$$

For each constraint θ_j that has already been scored on problem instance z, we define the constraint-wise upper bound as

$$\begin{aligned} B_{c,j}(\theta_i, z) =& J^z(\theta_j) + E_z[J^z(\theta_i) - J^z(\theta_j)] \\ &+ C_{2,i} \cdot \sigma_z[J^z(\theta_i) - J^z(\theta_j)] \end{aligned} \tag{2}$$

The intuition here is that if the two constraints had a consistent difference, then we can estimate the upper bound of the unevaluated constraint, θ_i, by adjusting the value of the evaluated constraint by that difference and its variance. From now on, we will refer to $C_{1,i}$ and $C_{2,i}$ as C values.

At the k^{th} round of evaluation, we would have evaluated $k-1$ constraints. Let

$$\tilde{\Theta} = \{\theta^{(1)}, \cdots, \theta^{(k-1)}\}$$

be the set of evaluated constraints so far, and let

$$B_{c,\tilde{\Theta}}(\theta_i) := \{B_{c,\theta^{(1)}}(\theta_i, z), \cdots, B_{c,\theta^{(k-1)}}(\theta_i, z)\}$$

be the set of constraint-wise upper confidence bounds on unevaluated θ_i. After k round of evaluations, we integrate all the upperbounds constructed so far, and select the one that has the tightest value as follows.

$$B_k(\theta_i) := \min\{B_c(\theta_i, z), B_{p,\tilde{\Theta}}(\theta_i)\} \tag{3}$$

Unfortunately, constructing the upper bound (3) with (1) and (2) is impossible since we do not have infinite samples in the

real world. Therefore, we have following finite approximations of these two equations. Using matrix $\mathfrak{J}$, we have,

$$B_p(\theta_i) \approx \mu(J^z(\theta_i)) + C_{1,i} \cdot s(J^z(\theta_i)) \quad (4)$$

$$B_{c,\theta_j}(\theta_i, z) \approx J^z(\theta_j) + \mu(J^z(\theta_i) - J^z(\theta_j)) + C_{2,i} \cdot s(J^z(\theta_i) - J^z(\theta_j)) \quad (5)$$

where μ and s denote sample mean and variance. Notice that (5) is a row-wise mean and variance, and (4) is a mean and variance of column-wise differences in matrix $\mathfrak{J}$.

We provide the detailed pseudo-code of our algorithm, BOX (Blackbox Optimization with eXperience) in Algorithm 1. The inputs to the algorithm include the C values for all the constraints, k, the number of rounds, Θ, the library of solution constraints, and lastly their respective scores in training problem instances, $\mathfrak{J}$.

In order to create the training data $\mathfrak{J}$ and Θ, we first call Algorithm 2 before calling BOX. This algorithm takes as inputs n, the number of training problem instances, u, the number of constraints to be generated for each problem instance, and the optional parameter Θ, which can be either be generated by the algorithm or passed in by external source. When Θ is not given, the algorithm runs *Solver* to generate a solution constraint. Solver is what you would do if you did not have the library of constraints. For instance, if our solution constraint is a robot base pose to pick an object up from a table, Solver would generate a solution constraint by randomly sampling from a continuous collision-free region in the space where the robot can reach the object, and check if there is a feasible path to pick the object. The algorithm generates u number of constraints for each problem instance, and we have in total $m = u \cdot n$ number of constraints. The algorithm then produces the matrix $\mathfrak{J}$ by scoring the plans associated with the constraints, using the *ScoreSolutionConstraint* function which returns the plan associated with the given constraint, and its score. The plan returned at this stage is ignored.

Once we have generated $\mathfrak{J}$ and Θ, we call BOX to choose the optimal constraint for the current problem instance z. BOX chooses $\theta^{(t)}$ using the upper bounds previously defined before, and evaluates it using the *ScoreSolutionConstraint* function which returns the numerical quality of the plan generated with $\theta^{(t)}$, and the plan, $P_{\theta^{(t)}}$, associated with the constraint. It repeats these two steps for k number of rounds. After k rounds, BOX returns the plan that has the highest score.

Algorithm 1 BOX($\{C_{1,i}\}_{i=1,\ldots,m}, \{C_{2,i}\}_{i=1,\ldots,m}, k, \Theta, \mathfrak{J}$)

$\tilde{\Theta} = \{\}$ *// a set of evaluated constraints*
$\tilde{\mathfrak{J}} = \{\}$ *// values of evaluated constraints*
$P_{\tilde{\Theta}} = \{\}$ *// plans of evaluated constraints*
Setup $\{B_c(\theta_i, z)\}_{i=1,\cdots,m}$ according to eqn. 5
$\theta^{(1)} = arg\min_{\theta \in \Theta} \mathfrak{B}_c$, break ties randomly
$\tilde{\Theta} = \tilde{\Theta} \cup \theta^{(1)}$
for $t = 2$ **to** k **do**
  Setup $\{B_{c,\tilde{\Theta}}(\theta_i, z)\}_{i=1,\cdots,m}$ according to eqn 4
  Setup $\{B_k(\theta_i)\}_{i=1,\cdots,m}$ according to eqn 3
  $\theta^{(t)} = arg\min_{\theta_i \in \Theta}\{B_k(\theta_i)\}_{i=1,\cdots,m}$, break ties randomly
  $[J^z(\theta^{(t)}), P_{\theta^{(t)}}] = ScoreSolutionConstraint(\theta^{(t)})$
  $\tilde{\mathfrak{J}} = \tilde{\mathfrak{J}} \cup J^z(\theta^{(t)})$
  $\tilde{\Theta} = \tilde{\Theta} \cup \theta^{(t)}$
  $P_{\tilde{\Theta}} = P_{\tilde{\Theta}} \cup P_{\theta^{(t)}}$
end for
$\hat{\theta}^* = arg\max_{\theta \in \tilde{\Theta}} \tilde{\mathfrak{J}}$
return $P_{\hat{\theta}^*}$

Algorithm 2 GenerateTrainingData($n, u, \Theta = \{\}$)

$z_1, \ldots, z_n \sim P(z)$ *// sample n problem instances*
// Generate solution constraints if not given
if Θ is empty **then**
  for $j = 1$ **to** u **do**
    $\theta_j = Solver(z_i)$
    $\Theta = \Theta \cup \theta_j$
  end for
  $m = n \cdot u$
else
  $m = |\Theta|$
end if
// Evaluate the constraints
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    $[J^{z_i}(\theta_j), \sim] = ScoreSolutionConstraint(\theta_j)$
  end for
end for
$\mathfrak{J} = AsMatrix(\{J^{z_i}(\theta_j), i = 1 \ldots n, j = 1 \ldots m\})$
return $\mathfrak{J}, \Theta, m$

III. Experiments

To evaluate BOX, we compare against four other solution-constraint -library-based algorithms in various challenging domains. The first one is random sampling, where we uniform sample k number of solution constraints. The second one is a regular-DOO, which uses Euclidean distance metric to estimate the upper bound of the constraints, instead of the correlation, the eqn (5). This algorithm is much motivated by [1], which was originally invented for black box function optimization problems. The third one is static ordering algorithm, which orders the grasps according only their expected values (i.e. eqn (3) without B_c) The last algorithm is Solver - this is what you would do if you did not have any of these library-based algorithms to solve the planning problem. We use RRT seeded with a fixed randomization seed value as our motion planner throughout experiments.

To evaluate the speed and quality of plans of these algorithms, we consider three different plots. In the first plot, we primarily compare the planning speed of these algorithms by looking at how fast these algorithms produce the first feasible plan. In the second plot, we look at, given different multiples of the time required by Solver to generate a plan, how much improvements in quality of plans that these library-based algorithms can provide. Lastly, we look at how long these algorithms to take to find the best solution constraint for the given planning problem instance.

To produce these plots, we run Leave One Out Cross Validation (LOOCV) on the training data. That is, we assume that we have $n-1$ training problem instances, and the one that is left out is used as a test problem instance. This

is repeated for all training instances, and the average of n different LOOCV results are reported.

A. Grasp selection for robust motion planning

In this experiment, the robot needs to plan its left hand motion from its initial pose to a pregrasp pose. To do this, the robot first needs to choose a grasp, perform inverse kinemactics to compute the pregrasp configuration, and then call a motion planner to compute a path. We would like to choose a grasp such that the planner, when constraint to plan to the given pregrasp pose, maximizes the distances to the obstacles along its trajectory. Therefore, here, our solution constraints are various grasps.

A planning problem instance for this domain is defined by an arrangement of the objects on a table. Figure 2 shows two instances of this problem, which are also part of the training data. We have four different obstacles including two mugs, two boxes, and the object to be picked up (the blue object). To generate a planning problem instance, we either remove an object with probability 0.5, or randomly change its location on the table. The blue object can only be moved. The robot's active degrees of freedom (DOF) are its left arm and torso, summing up to 8 degrees of freedom. Other DOFs remain fixed. The robot's initial configuration remains fixed across different problem instances as well.

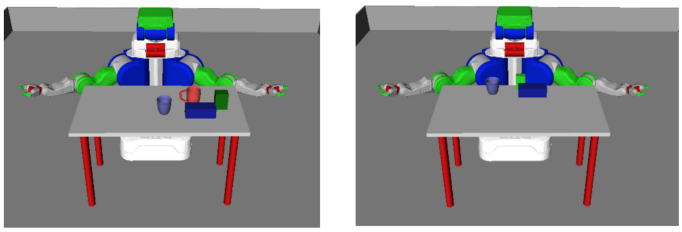

Fig. 2: Two different instances of grasp selection domain. The robot's objective is to plan a its left hand motion to a pregrasp pose for the blue object. The arrangement and existence of obstacles are random across different planning problem instances.

The goal for this experiment is to select a grasp that can be reached "robustly", that is, the grasp approach motion is collision-free under control uncertainty and/or uncertainty in poses of objects. We measure robustness as the average (over the obstacles) of the minimum distance to each obstacle, when we plan a path to the selected grasp. That is, let the plan $P_\theta := \{c_1 = c_{init}, c_2, \ldots, c_{n-1}, c_n = c_\theta\}$, be a path of robot's left arm and torso, which consists of a sequence of robot configuration from initial configuration c_{init}, to the pregrasp configuration, c_θ. Then, the function that we would like to maximize is

$$J^z(\theta) = \begin{cases} \frac{\sum_{i=1}^{n} min\_dist\_to\_obs(c_i)}{n} & \text{if } \theta \text{ feasible} \\ -0.0719, & \text{otherwise} \end{cases} \quad (6)$$

where min operator over matrix $\mathfrak{J}$ returns the minimum element of the matrix, and the *avg* operator takes the average of the elements in the matrix. *min_dist_to_obs* function calculates the distance to the closest obstacle from the robot at each node of the planned trajectory.

We consider 121 different grasps, computed with OpenRAVE grasp model function, with 479 training problem instances. We pass these precomputed 121 grasps as Θ to BOX. The training data were generated by trying all the grasps in each of 479 scenes, and recording the score of the motion that the RRT generates. Our *ScoreSolutionConstraint* function works as follows. Given grasp, it checks whether there exists a collision-free inverse kinematics. If the grasp fails to find one, then we give the grasp a reward of zero. Otherwise, we call RRT to plan a motion. If RRT fails to find a path, then the grasp receives a reward of zero. If it succeeds in finding a path, then the path gets scored using the function (6).

Solver for this domain is a random sampler that tries a grasp without replacement until a feasible one is found. Unlike the random sampling algorithm, this one stops searching as soon as a feasible grasp is found instead of sampling k grasps. Solver has a time limit of 120 seconds. If it cannot find a grasp within 120 seconds, then it receives a reward of 0.

The inputs to BOX were: $n = 479, C_{1,i} = 10, C_{2,i} = 2, i = 1, \ldots, m, \Theta = openrave\_grasps$. Figure 3 shows the comparison of algorithms for average time for finding the first feasible plan for each problem instance. Solver, which randomly samples a grasp without replacement, takes close to 6 times that of the BOX. RegDOO and static algorithms are much more slower than BOX, each requiring multiple factor of the time required by BOX.

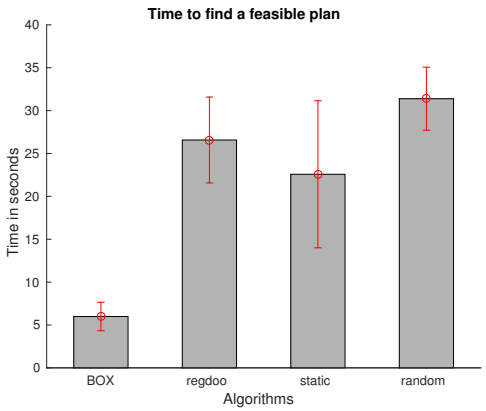

Fig. 3: Average time required by each algorithm over all problem instances to find a feasible plan with 95% confidence interval.

B. Grasp and base location selection for robust motion planning

We again tackle the robust motion planning problem in a more challenging domain. In this experiment, the robot again needs to plan its left arm and torso to reach a pregrasp

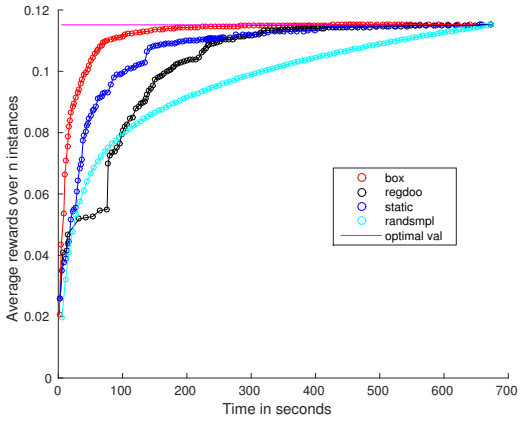

Fig. 4: Grasp domain

pose, but it also needs to place its base location appropriately in addition to choosing a grasp in order to maximize the distances to obstacles. Therefore, our a solution constraint for this domain consists of the robot base pose, (x, y, ψ), where ψ is an orientation of the robot, as well as one of 121 grasps. The objective function we would like to maximize is same as eqn. (6)

A planning problem instance is again defined by the arrangement of objects. Figure 5 shows three different training scenes. We have 20 rectangular boxes as obstacles, all standing up on the two tables. The (x, y) location and orientation about $z-axis$ of each of the obstacles are randomly chosen. The object to be picked up, colored in blue, is lying on either one of the two tables at random, with random orietntation about $z-axis$ and (x, y). The robot always starts at the same base pose, with same arm configuration. The robot has 3 active DOFs when it is required to move its base to a pick-up base pose. Once it gets there, the robot then has 8 active DOFs, including its left arm and torso. We omit the base motion planning and only consider arm motion for evaluation.

Given a planning problem instance, Solver for this domain performs three procedures. First, it randomly samples a base pose, (x, y, ψ), from a continuous, reachable and collision free region of the space. It then tests 121 grasps sequentially until a collision free inverse kinematics is found. Lastly, it calls RRT at this base pose, to the collision free pregrasp configuration, and see checks if the path exists. If there is, then this pair of the chosen grasp g and the robot pose (x, y, ψ) becomes a solution constraint, and is added to our library. Once we have constructed the library, for each problem instance and a constraint, *ScoreSolutionConstraint* function then calls RRT using the constraint, and then score them according to eqn. (6), with the score of -0.3927 for the infeasible constraints. However, if the base pose is in collision, or if a reachable collision-free pick-up base location is found but none of the grasps have collision-free inverse kinematics, or RRT fails to find a path, the constraint is assigned a score of 0.

The inputs to BOX were $n = 988, u = 1, C_{1,i} = 10, C_{2,i} = 1, i = 1, \ldots, m, \Theta = \{\}$. We then discarded some of the generated solution constraints and reduced the $|\Theta| = 486$, to speed up the training data generation. [1]. Figure 6 shows the average time for algorithms to generate the first feasible plan. Unlike

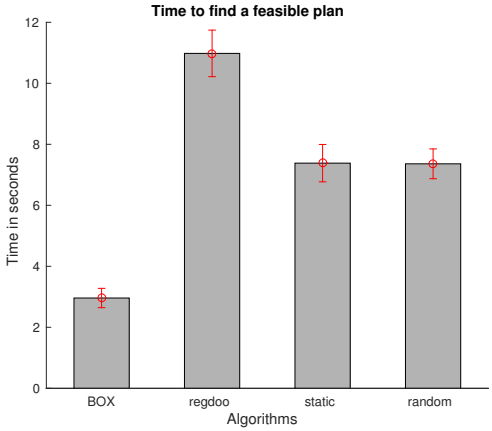

Fig. 6: Average time required by each algorithm to find a feasible plan for a problem instance.

the previous domain, the library-based algorithms completely outperformed Solver. This is because compared to computing the inverse kinematics from sampled grasp and base pose and planning the left arm motion, sampling a feasible θ takes significantly more time. Out of all the library based methods, BOX again outperformed all the other algorithms by multiple factors.

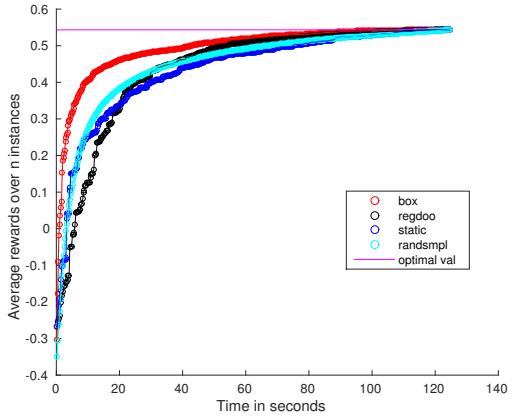

Fig. 7: pick-base domain.

C. Grasp, base location, object placement pose, and partial path selection for shortest path planning with varying robot shape

In this experiment, the robot needs to pick-and-place an object from one table to another, where there is a narrow

[1]This step is completely unnecessary if time for training data generation is not of a concern

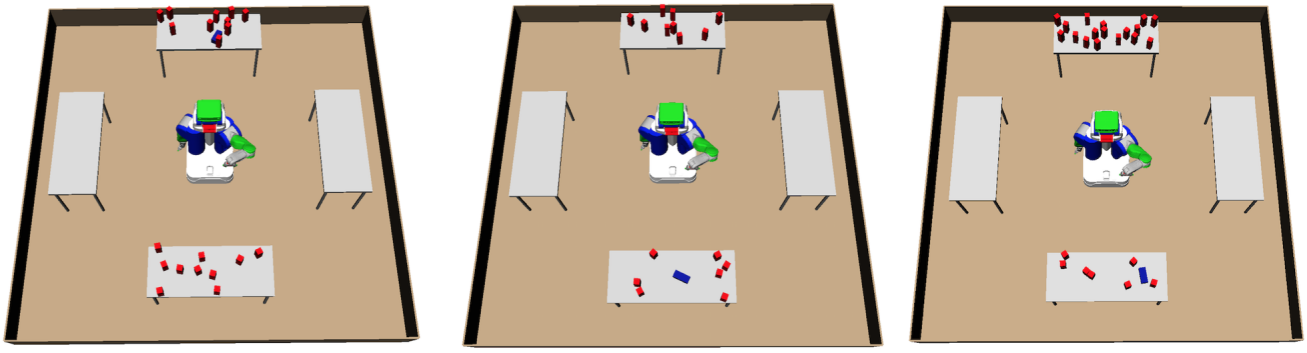

Fig. 5: Three different instances of base and grasp selection domain, with robot at its default location. The robot needs to choose its base pose and a grasp to plan a motion to a pregrasp configuration for the blue object. The poses of obstacles and the blue object is random.

passage in between, using its left arm, torso, and base motions. To do this, the robot first needs to choose three variables: a grasp, g, an object placement pose, p_{obj}, which consists of (x,y) location on the other table and its orientation about the $x-axis$, a robot base pose, p_{robot}, for placing the object at p_{obj}. Once the robot chooses these variables, it calls RRT to generate three paths: the pregrasp configuration, a path from the initial base pose to p_{robot}, and then a path for placing the object at p_{obj}. In addition to forementioned three variables, we would like to suggest subgoals for each of these paths. Therefore, a solution constraint in this domain would consists of six variables, $[g, p_{obj}, p_{robot}, sg_{pregrasp}, sg_{base}, sg_{place}]$, where sg denotes the subgoal for each path. The objective here is to minimize the total distance travelled by the robot's leftarm, torso, and base.

Unlike the previous two experiments where a problem instance was defined only by arrangements of obstacles, here a problem instance is defined by both the obstacle arrangement and the length of the object to be transported. We have in total 28 rectangular obstacles, all standing up, whose (x,y) locations on either table is chosen randomly. The initial pose of the object to be transported remains the same across different problem instances, while its length is chosen randomly from three different values. Figure 8 shows three different training scenes. Here, when the robot is holding the black object with different lengths, this effectively changes the shape of the robot, and it needs different maneuvers to get through the passage for different shapes. More specifically, since the longest rod has length greater than the width of the passage, the robot needs to turn its base in order to get through it. The robot has 8 active DOFs for planning its left arm and torso motions for pregrasp and placing, and 3 active DOFs for planning its base pose to p_{robot}.

Given a planning problem instance, Solver first randomly samples g, p_{obj}, p_{robot} by first finding the feasible grasp, then choosing the p_{obj} for which inverse kinematics yields the g. Then, it samples collision free p_{robot} for which this grasp is reachable. Once it finds chose these variables, it, then

Fig. 8: Three different instances of the pick-and-place domain. The robot needs to transport the black object, whose length is randomly chosen from the three lengths shown, to the other table. The obstacle arrangements across different scenes are random as well. The initial pose of the black object is the same across all planning problem instances.

it calls RRT for computing three different paths: a pickup motion, a base motion, and a place motion. To construct a library, we then choose, as our subgoals for these paths, the configuration that minimizes the distance travelled by the robot when planned from the initial configuration to the subgoal then to the goal pose. If all these are feasible, we then add it to the library, otherwise discard it.

Once we have constructed the library, for each problem instance and a constraint, *ScoreSolutionConstraint* function calls RRT with the constraint and score the computed motion

with the function

$$J^z(\theta) = \begin{cases} -length(P_\theta), & \text{if } \theta \text{ feasible} \\ \min(\mathfrak{J}) - |avg(\mathfrak{J})|, & \text{otherwise} \end{cases} \quad (7)$$

where min operator over matrix $\mathfrak{J}$ returns the minimum element of the matrix, and the *avg* operator takes the average of the elements in the matrix.

The inputs to BOX were: $n = 600, u = 1, C_1 = 10, C_2 = 2$, and we discarded some of θ so that $|\Theta| = 586$. Figure 9 shows the average time for finding the first feasible plan. This domain again required Solver to sample a value from continuous domain, such as p_{obj} and p_{robot}, which gave the library-based approaches more advantage. Moreover, the narrow passage made the plain RRT, without subgoal, to have more planning time than those with subgoal. These two advantages allowed the library based algorithms to compute a plan faster, with BOX as fastest algorithm, with more than 4 times faster than Solver. Static came next, which was more than 2 times faster than Solver. Random sampler struggled the most, with its speed almost same as Solver.

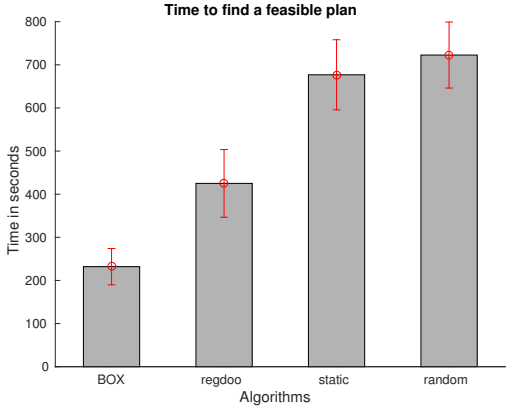

Fig. 9: Average time required by each algorithm to find a feasible plan for each problem instance.

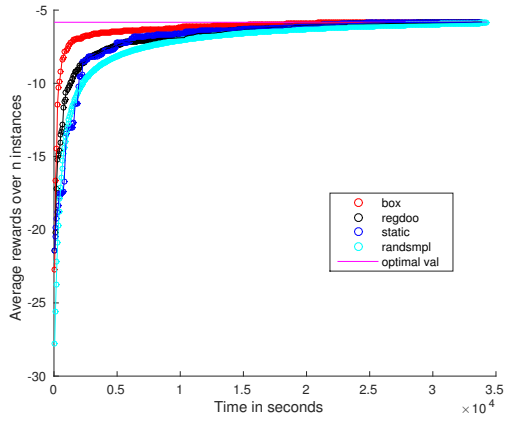

Fig. 10: Biggest domain.

REFERENCES

[1] R. Munos. Optimization of deterministic functions without the knowledge of its smoothness. *Advances in Neural Information Processing Systems*, 2011.