Multi-Object Rearrangement with Monte Carlo Tree Search: A Case Study on Planar Nonprehensile Sorting

Haoran Song[1]*, Joshua A. Haustein[2]*, Weihao Yuan[1],
Kaiyu Hang[3], Michael Yu Wang[1], Danica Kragic[2], Johannes A. Stork[4]

***Abstract*— In this work, we address a planar non-prehensile sorting task. Here, a robot needs to push many densely packed objects belonging to different classes into a configuration where these classes are clearly separated from each other. To achieve this, we propose to employ Monte Carlo tree search equipped with a task-specific heuristic function. We evaluate the algorithm on various simulated sorting tasks and observe its effectiveness in reliably sorting up to 40 convex objects. In addition, we observe that the algorithm is capable to also sort non-convex objects, as well as convex objects in the presence of immovable obstacles.**

I. INTRODUCTION

Rearranging objects, e.g., to clear a path or clean a table, is an essential skill for autonomous robots. For this, a robot needs to plan ahead in which order it should move the objects and whereto. This rearrangement planning problem is known to be NP- or even PSPACE-hard depending on the goal definition [1]. Accordingly, various specialized algorithms have been proposed that address specific practical rearrangement problems efficiently. For instance, several prior works specifically address navigating a mobile robot among movable obstacles (NAMO) [2–5]. Similarly, clearing clutter for grasping has been addressed by pushing obstacles aside locally [6–8], or recursively removing obstructions through pick-and-place [9]. Even for large-scale rearrangements, where many objects need to be arranged to target locations, efficient approximative algorithms have been proposed. While early works [9, 10] were limited to *monotone* problems, where each object needs to be moved at most once, recent works have overcome this limitation [11–15]. Large-scale rearrangements, however, have predominantly been addressed using pick-and-place or single-object pushing.

We are interested in large-scale *non-prehensile* rearrangement problems, where a robot pushes multiple objects simultaneously to reach a goal that is characterized by the final poses of many objects. Specifically, we consider the planar non-prehensile sorting task illustrated in Fig. 1. Here, a planar pusher has to sort objects belonging to different classes into homogenous distinct clusters. The problem is algorithmically challenging, as it is non-monotone, requires multi-object pushing to be solved efficiently, and the robot needs to circumnavigate obstacles.

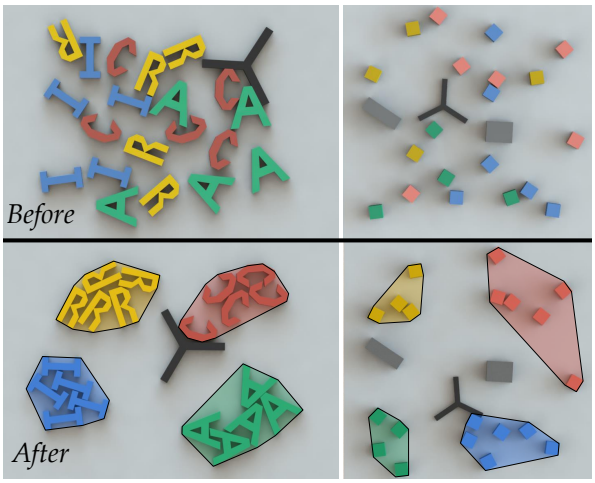


Fig. 1: The planar push sorting task: A planar robot (black) is tasked to separate objects belonging to different classes into homogenous distinct clusters, optionally in the presence of obstacles (grey).

We propose to address this problem using Monte Carlo tree search (MCTS) [16]. Monte Carlo tree search is a planning algorithm for sequential decision making problems, and is well suited for this sorting task. First, the algorithm can search high-dimensional state spaces by only performing a forward search. This allows us to employ commonly available physics models to predict the outcome of pushing actions that involve complex multi-object, multi-contact dynamics. Second, the algorithm employs an adaptive sampling strategy that focuses its search on the parts of the state space that are relevant to solving the problem. This is particularly important as modeling multi-contact physics is computationally expensive. In fact, we observe that the algorithm is efficient enough to replan after each push, allowing the system to compensate for errors in the physics propagation. Third, MCTS requires no explicit target states, but instead can be applied when there is only a discriminative function to evaluate whether a state is a goal. This is the case in the sorting task, where the goal is defined through relative positions of objects rather than absolute target positions.

The contribution of this work lies in adapting this algorithm to the sorting task, and evaluating it on a variety of scenarios. To reduce the need for long physics rollouts,

[1]These authors are with the Robotics Institute, Hong Kong University of Science and Technology, Hong Kong, China. H. Song is with the Department of Mechanical and Aerospace Engineering. W. Yuan is with the Department of Electronic and Computer Engineering. M. Y. Wang is with the Department of Mechanical and Aerospace Engineering and the Department of Electronic and Computer Engineering.

[2]J. A. Haustein and D. Kragic are with the Centre for Autonomous Systems, EECS, KTH Royal Institute of Technology, Stockholm, Sweden.

[3]K. Hang is with the Department of Mechanical Engineering and Material Science, Yale University, New Haven, Connecticut, USA

[4]J. A. Stork is with the Centre for Applied Autonomous Sensor Systems, Orebro University, Orebro, Sweden.

*These authors contributed equally to this work

we propose a heuristic function that successfully guides the algorithm towards sorted states. In addition, inspired by the recent striking success of AlphaGo [17], we also train a roll-out policy from data to improve the algorithm's performance further. We evaluate the approach for different numbers of objects and classes, different object shapes (convex and non-convex), and sorting in the presence of immovable obstacles.

The remainder of this paper is structured as follows. We first formally define our sorting task in Sec. II, before discussing related work in more detail in Sec. III. Thereafter, we provide background information on MCTS in Sec. IV and present our adaptations in Sec. V. We present experimental results in Sec. VI and conclude in Sec. VII.

II. PROBLEM DEFINITION

In the planar push sorting problem (PPSP) a robot $\mathcal{R}$ is tasked to sort a set of movable objects $\mathcal{M}$ in a bounded workspace according to a given class membership, see Fig. 1. The workspace is planar and all objects are assumed to be rigid. Accordingly, the state spaces are $\mathcal{X}_i \subset SE(2)$ for $i \in \{\mathcal{R}\} \cup \mathcal{M}$, and the composite state space of the world is $\mathcal{X} = \mathcal{X}_1 \times \ldots \mathcal{X}_{|\mathcal{M}|} \times \mathcal{X}_{\mathcal{R}}$. The workspace may contain immovable obstacles $\mathcal{O}$ that need to be avoided. Accordingly, we refer to states $x \in \mathcal{X}$ to be valid, if there are no two objects intersecting and no collisions with obstacles.

Each movable object belongs to exactly one class $c \in \mathcal{C}$. These classes are user-defined and can be based on, for example, shared physical properties (e.g., color, shape, size) or a common functional purpose of the objects. The task of the sorting problem is to rearrange the objects into a *sorted* valid state according to their class membership. A *sorted* state is a state where the objects of each class form disjoint clusters, see Fig. 1. More formally, let $\mathrm{CH}(c_i, x) \subset \mathbb{R}^2$ denotes the smallest convex set that contains all objects of class $c_i \in \mathcal{C}$ in state $x \in \mathcal{X}$. Furthermore, let the distance function

$$d_c(A, B) = \begin{cases} \min_{a \in A, b \in B} \|a - b\|_2 & \text{if} A \cap B = \varnothing \\ 0 & \text{else.} \end{cases} \quad (1)$$

denote the smallest pairwise distance between elements of two sets, $A, B \subset \mathbb{R}^2$. We define a state $x \in \mathcal{X}$ to be *sorted*, if all classes have at least a distance $\epsilon > 0$ from each other and the obstacles:

$$\begin{aligned} \texttt{Sorted}(x) : \Longleftrightarrow & \min_{\substack{i,j \in \mathcal{C} \\ i \neq j}} d_c(\mathrm{CH}(i, x), \mathrm{CH}(j, x)) > \epsilon \\ & \wedge \min_{i \in \mathcal{C}} d_c(\mathrm{CH}(i, x), \mathcal{O}) > \epsilon. \end{aligned} \quad (2)$$

Let $\mathcal{A}$ be the set of planar motions that $\mathcal{R}$ is able to execute. We limit $\mathcal{A}$ to motions that are sufficiently slow, so that pushing dynamics can be assumed to be quasistatic [18]. Given an initial valid state $x_0 \in \mathcal{X}$, the problem of planar push sorting is then to compute and execute a series of actions $a \in \mathcal{A}$ that transfer the system from state x_0 to any valid sorted state $x_g \in \mathcal{X}$. In this process, all intermediate states x_i have to be valid, i.e., not colliding with any immovable obstacle or be out of bounds.

We consider this problem in scenarios, where objects are initially densely packed. In addition, objects of the same class eventually need to be pushed into the same region. This renders the ability to purposefully push multiple objects simultaneously essential to efficiently solve this task.

III. RELATED WORK

In this section, we first discuss prior works on non-prehensile rearrangement that exploit multi-object pushing. Thereafter, we discuss prior applications of Monte Carlo tree search in the context of rearrangement planning.

A. Non-prehensile Rearrangement

Non-prehensile rearrangement covers a variety of different tasks. We distinguish between navigation or manipulation among movable obstacles, and large-scale rearrangement tasks. In navigation or manipulation among movable obstacles the priority is to navigate the robot or transport individual objects in the presence of clutter. In other words, the goal is expressed with respect to a few individual objects or the robot, while the remaining objects may be placed anywhere. This category includes repositioning tasks [19–25] reaching for an object within clutter [6–8, 26, 27], as well as singulating or separating individual objects [28–31]. By large-scale rearrangement we refer to problems where the goal is expressed in terms of many objects, and all final poses are relevant to the task. Our sorting task is such a problem, and to the best of our knowledge only Huang et al.[15] have previously addressed such problems in combination with multi-object pushing.

Huang et al. found that iterative local search (ILS) equipped with strong heuristics and an ϵ-greedy rollout policy succeeds at solving various table-top rearrangement tasks, including a sorting task of up to 100 cubes. The addressed sorting problem, however, differs from ours in two key aspects. First, for the sorting goal, explicit target locations for each class are provided as input. This allows to derive a heuristic for action sampling by computing the direction in which each object should be pushed. In our problem, in contrast, no explicit target locations are provided, and instead the planner needs to select suitable locations to achieve a sorted state itself. Second, the problem specifically addresses table-top sorting with a manipulator that is capable of moving the pusher in and out of the pushing plane at any location. In our problem, the pusher's motion is constrained to the pushing plane, requiring it to circumnavigate objects and obstacles. Designing a strong rollout policy for such navigation tasks—as needed for ILS— is non-trivial. MCTS, in contrast, does not require similar heuristics, and succeeds even with a random rollout policy.

The additional challenges of our sorting problem are useful for two reasons. First, relieving the user from providing a sorted target state as input makes the algorithm easier to use. Second, constraining the pusher's motion to the plane is more general. It applies to mobile robots, as well as to manipulators that have few degrees of freedom, such as Delta robots. In addition, although not explicitly studied in this

work, our problem formulation resembles a sorting task in constrained spaces such as shelves, and may provide relevant insights for future work in this direction.

B. Monte Carlo Tree Search for Rearrangement Planning

Monte Carlo tree search has recently been applied to rearrangement planning problems using pick-and-place. Zagoruyko et al.[32] use MCTS to rearrange up to 9 objects to user-given target pose. The authors also train a rollout policy from solutions produced by MCTS that makes the algorithm efficient enough to replan online and thus compensate for disturbances during the execution.

King et al. [33] proposed to apply MCTS for pushing a single object among movable obstacles under uncertainty. This approach focuses on computing the most robust sequence of pushing actions by planning on belief space. Here, the adaptive sampling of MCTS makes this tractable by focusing the limited computational budget for constructing a state belief model only on the most promising trajectories.

In contrast, we employ MCTS on a large-scale non-prehensile sorting task, and address uncertainty only indirectly through replanning. For this, we equip the algorithm with a heuristic function that allows us to limit the computationally expensive rollouts, making the algorithm efficient enough for replanning after each push.

IV. BACKGROUND

In this section, we describe the basic form of Monte Carlo tree search (MCTS) [16], which we use in our planar push sorting planner described in Sec. V. MCTS is used for sequential decision making problems with state space $\mathcal{X}$, action space $\mathcal{A}$, reward signal g, and transition model Γ. Given a current state $x_t \in \mathcal{X}$ the algorithm estimates the state-action value function $q(x_t, .)$ using simulated episodes called rollouts and returns the best action. During rollout, it uses a (often simple) rollout policy to decide on actions and simulates state transitions using Γ. To focus on high-reward regions, MCTS builds a search tree with states as nodes and actions as edges, rooted at the current state. With this tree, the algorithm maintains value estimates for the states that are most likely to be reached within a few steps. For every single iteration, the algorithm executes the following steps which are also shown in Fig. 2:

Selection: Use a tree policy π_{tree} to traverse from the root to a leaf node. The tree policy exploits the state-action value estimates for the states in the tree and balances exploration and exploitation.

Expansion: Expand the search tree by selecting an action and adding the reached state as a child node.

Simulation: Use the rollout policy π_{roll} for action selection and simulate the episode until termination according to Γ.

Backup: Use the return generated by the episode to update the state-action value estimated for the traversed edges in the search tree.

Often, MCTS is set to terminate after a certain number of iterations $n_{\max}$ or through some statistical criterion. Nodes within the tree are first fully expanded before any of their children are expanded. If terminal states are too many steps away for full rollout simulation to be tractable—as it is in our case—truncated rollouts are used. Then, instead of the return of the complete episode, some heuristic estimate of the return is backed up. Due to the policy improvement theorem, selecting the best action at the root node is at least as good as the rollout policy. However, because the value estimates are based on long-term consequences, it is usually better. The details of how we adapt MCTS to the planar push sorting problem are explained in the following section.

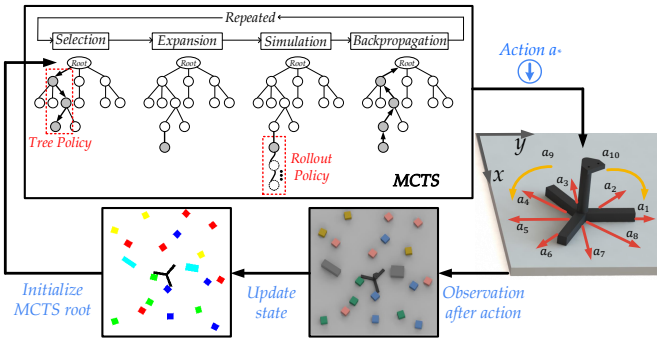


Fig. 2: The sorting planner's scheme for real-world execution.

V. PLANAR PUSH SORTING PLANNER

In this section, we first model the PPSP as sequential decision-making problem for MCTS. Then, we outline how our algorithm uses MCTS and decides on termination. We describe how we simulate PPSP in Monte Carlo rollouts and detail the four MCTS steps mentioned in Sec. IV. Finally, we explain how we use deep learning to obtain a rollout policy from data to improve our MCTS.

A. Sequential Decision-making Problem

To model the PPSP we use the full configuration space $\mathcal{X}$ as the state space and define a robot-centric action space $\mathcal{A}$ with 10 actions as depicted in Fig. 2 on the right. The robot can translate into 8 different directions and rotate left and right in small increments. As transition model Γ we employ the physics simulator Box2D[1], which is capable of modeling multi-object interactions. We model the PPSP as a deterministic process, and compensate for errors in the physics modelling by replanning after each push.

With our action space, it often takes up to 200 transitions until a sorted state is reached, which makes large numbers of full rollouts with the physics simulation practically intractable. For this reason, we use truncated rollouts of length $d_{\max}$. However, this means that we cannot use $\texttt{Sorted}(x)$ from Eq. (2) as a feedback signal since most rollouts do not reach a sorted state. Therefore, we define a different reward signal $g(x)$ that provides useful feedback also for unsorted states and increases when the state becomes more sorted.

Reward Signal: We construct the reward signal for a state x from four components: A measure how compact a class c_i is, $E_i^{\text{self}}(x)$, how far away class centers are from each other, $E_{ij}^{\text{other}}(x)$, how far away class centers are from obstacles,

[1]Box2D, A 2D Physics Engine for Games: `https://box2d.org/`

$E_i^{\text{obst}}(x)$, and the distance of the two closest class centers, $d_{\text{cent}}(x)$. Concretely, we define

$$E_i^{\text{self}}(x) = \frac{1}{|c_i|} \sum_{m \in c_i} \ln(p_i(x_m)) \tag{3}$$

$$E_{ij}^{\text{other}}(x) = \ln(1 - p_i(\mu_j(x))) \tag{4}$$

$$E_i^{\text{obst}}(x) = \sum_{o \in \mathcal{O}} \ln(1 - p_i(\mu_o)) \tag{5}$$

$$d_{\text{cent}}(x) = \min_{i,j \in \mathcal{C}, i \neq j} \|\mu_i(x) - \mu_j(x)\|_2, \tag{6}$$

where $\mu_i(x)$ denotes the mean position of all objects in class c_i, μ_o the centroid of obstacle o, x_m the position of object m, and $p_i(x_m) = e^{-\lambda(\|x_m - \mu_i\|_2^2)}$ a Gaussian with variance λ centered at the mean position of class c_i. The term $E_i^{\text{self}}(x)$ increases as objects of class c_i are more compact. The term $E_{ij}^{\text{other}}(x)$ increases as the centers of class c_i, c_j are moved apart. Similarly, the term $E_i^{\text{obst}}(x)$ increases as the center of class c_i is moved away from obstacles.

We combine these terms to a reward signal

$$g(x) = \frac{\sum_{i=1}^{|\mathcal{C}|} (E_i^{\text{self}}(x) + \sum_{j=1}^{i-1} E_{ij}^{\text{other}}(x) + E_i^{\text{obst}}(x))}{d_{\text{cent}}(x)}. \tag{7}$$

It is $g(x) < 0$ for all states, but it approaches 0 as members of the same class get closer and members of different classes separate. It eventually reaches higher values for sorted states (as defined in Eq. (2)) than for unsorted states. Hence, by maximizing $g(x)$ our planner will gradually aggregate objects of the same class and separate those from different classes.

B. Sorting Planner Outline

Starting with an unsorted state x, our sorting algorithm repeatedly runs MCTS to obtain the best action for this state, executes the action and observes the result. This process is illustrated in Fig. 2 and Algorithm 1. In case we reach a sorted state, the sorting algorithm terminates. Due to the limited number of iterations of MCTS, the algorithm may become trapped in deep local optima. In order to detect such situations, the planner applies two strategies. First, it counts the number of subsequent actions where the robot has not pushed a single object, and terminates if this number exceeds a conservatively chosen threshold. Second, it compares the best reward observed for any visited state during any of the rollouts in MCTS, $\hat{g}$, with the reward of the current state $g(x)$. If the relative difference $\frac{\hat{g} - g(x)}{|g(x)|}$ is smaller than a small threshold $\nu > 0$, the planner also returns failure since there is no perspective of improving the state any further.

C. Monte Carlo Tree Search Implementation

We adapt the MCTS algorithm to our deterministic modeling and reward signal $g(x)$. The pseudo-code is shown in Algorithm 1. In the MCTS search tree, every node corresponds to a state $x \in \mathcal{X}$. Accordingly, the root node corresponds to the current state x_t. For each node s, the search tree stores the visit count $N(s)$, and estimates of an

Algorithm 1: Planar Push Sorting Planner

```
x ← Observe the state
while not Sorted(x) do
    a, ĝ ← MCTS(x)
    if IsTrapped(x, ĝ) then Terminate with Failure
    Execute(a)
    x ← Observe the state
Terminate with Success
function MCTS(x_0)
    Initialize the search tree with root x_0
    ĝ ← g(x_0); i ← 0
    for i < n_max do
        Select the root node
        while Node is fully expanded do
            Select child node according to π_tree
        Expand the current (leaf) node using π^0_roll
        g_max ← Truncated simulation rollout using π_roll
        Backup with return g_max
        ĝ ← max(g_max, ĝ)
    a* ← Greedy action selection for x_0
    return a*, ĝ
```

upper and lower bound of the reachable reward from its state, $\hat{V}_{\text{upper}}(x)$ and $\hat{V}_{\text{lower}}(x)$, respectively.

Selection and **Expansion** are executed as described in Sec. IV and we use the tree policy π_{tree} as defined below.

Simulation: We use the simulator Γ to predict the effect a robot action has as it pushes through objects. The actions are selected either from a random rollout policy π_{roll}, or from a learned rollout policy which is detailed below. The rollout is truncated after $d_{\max}$ steps to save computation. Since the rollout policy is sub-optimal—e.g. it may select actions which worsen previous gains—we return the maximal reward signal from the rollout $g_{\max}$ instead of the reward signal at the truncation state as a heuristic estimate of the episode's return.

Backup: When a rollout is finished, we increment the visitation counter $N(s)$ and update the bound estimates for each traversed node s in the search tree,

$$\hat{V}_{\text{upper}}(x) \leftarrow g_{\max}, \qquad \text{if } g_{\max} > \hat{V}_{\text{upper}}(x) \tag{8}$$

$$\hat{V}_{\text{lower}}(x) \leftarrow g_{\max}, \qquad \text{if } g_{\max} < \hat{V}_{\text{lower}}(x), \tag{9}$$

where x is the state for node s.

Tree policy: During selection, we use a tree policy π_{tree} that balances between exploration and exploitation and additionally considers the visitation count. For this we estimate the state-action value estimate $Q(x, .)$ at a node according to UCB1 [34] as

$$Q(x, a_i) = \frac{\hat{V}_{\text{upper}}(x_i) - \hat{V}_{\text{lower}}(x)}{\hat{V}_{\text{upper}}(x) - \hat{V}_{\text{lower}}(x)} + C\sqrt{\frac{2 \ln N(s)}{N(s_i)}} \tag{10}$$

where x_i is the state of the child node corresponding to action $a_i \in \mathcal{A}$. The exploration term is $C = \frac{1}{\sqrt{2}}$. We choose this tree policy as we model the decision process to be deterministic.

Rollout policy: While a completely random rollout policy guarantees probabilistic completeness, it is much more sample effective to select actions in an informed way during simulation. For this reason, we learn a rollout policy from

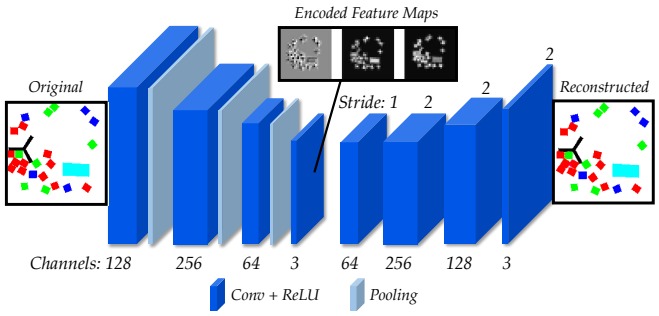


Fig. 3: Architecture of autoencoder.

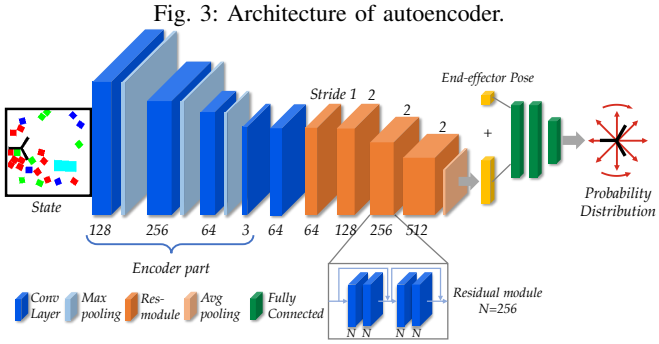


Fig. 4: Architecture of policy network.

successful sorting experience of a random rollout policy π^0_{roll} as detailed in Sec. V-D below.

D. Learning the Rollout Policy

The rollout policy π_{roll} has to map states $x \in \mathcal{X}$ to actions $a \in \mathcal{A}$ and, after the physical simulation, presents the second computational bottleneck during Monte Carlo rollout. In our case, the state space can also be different for different numbers of objects and obstacles. For this reason, we encode the positions of objects, obstacles and the robot in the state x as a 2D image color $\mathcal{I}(x) \in [0,1]^{256\times256\times3}$ which shows the footprint of each object colored by class. To get a more compact state representation, we learn a lower-dimensional embedding of this image space $f\colon \mathcal{I}(x) \mapsto \tilde{x} \in [0,1]^{32\times32\times3}$. For the rollout policy, we learn a mapping of the tuple $(x_{\mathcal{R}}, \tilde{x})$ to probability distribution over actions, P. Details are stated below.

State Representation Learning: We model the embedding function f as the encoder module in a convolutional auto encoder as seen in Fig 3. The training data is generated from our simulator, consisting of 120000 images with each image contains 20-30 square objects. The training objective is the mean squared reconstruction loss.

Rollout Policy Learning: We model the rollout policy $\pi_{\text{roll}}\colon (x_{\mathcal{R}}, \tilde{x}) \mapsto (P(a_1), \ldots, P(a_{10}))$ with the deep convolutional architecture ResNet-18, which is reported to be easier to optimize and resilient against overfitting [35], as depicted in Fig 4. The ResNet structure was initially designed for image classification while it is employed here for mapping the state feature to action labels. For the training data, we run our planar push sorting planner (see Algorithm 1) as described above with a fully random rollout policy π^0_{roll} for sorting 16, 20 and 24 cubes with one static obstacle randomly placed in the scenes. We record the observed transitions for each solved sorting problem as tuples $(\mathcal{I}(x), x_{\mathcal{R}}, Q(\mathcal{A}))$, where $Q(\mathcal{A})$ are the exploitation terms of the state-action value estimates for x in Eq. (10).

We train the model with cross-entropy loss while keeping the encoder parameters constant. The training target is the probability distribution over actions which arises from normalizing the state-action values, i.e. $P(a_i) = Q(a_i)/\sum_{a\in\mathcal{A}} Q(a)$.

VI. EXPERIMENTS

In this section, we investigate how our sorting algorithm performs in sorting 1) different number of objects and classes, 2) non-convex objects and 3) convex objects in the presence of immovable obstacles. We evaluate the performance of the algorithm with respect to success rate and number of actions in simulation. We employ the same physics model as Huang et al. [15], who observed good success rates on a real robot when replanning after each push. We therefore refrain from similar quantitative robot experiments here, and refer to the accompanying video for a qualitative demonstration on a real robot and simulation.

A. Experimental Setup

We use three types of objects for our evaluation: cubes of size $2.5cm \times 2.5cm$, U-shaped non-convex objects that can surround a cube, and randomly generated rectangular obstacles with an area no larger than $12.5cm^2$. All objects are randomly placed in a $50cm \times 50cm$ workspace. The pusher's action space is set to $5cm$ translations and rotations of $45°$. Accordingly, we set the maximal number of actions the robot is allowed to execute without contacting any object to 15. We run the first two sets of experiments with random rollout policy. In the third set, the most difficult one, we additionally evaluate the learned rollout policy.

Unless stated otherwise, all evaluations are run with the following parameters: $\epsilon = 0.05m$, $\nu = 0.05$ and $d_{\text{max}} = 3$. The number of iterations n_{max} for MCTS is dynamically adjusted. For this, we define two additional parameters n_{min} and ν_t. In each step of the sorting algorithm, MCTS is run for at least n_{min} iterations. If the best encountered reward value $\hat{g}$ in those rollouts does not sufficiently improve over the current state's reward, $\frac{\hat{g}-g(x)}{|g(x)|} < \nu_t$, the algorithm runs MCTS for additional n_{min} iterations. This is continued until either a sufficiently good $\hat{g}$ has been observed or n_{max} iterations have been reached. Unless stated otherwise, we set $n_{\text{min}} = 500$, $n_{\text{max}} = 5000$ and $\nu_t = 0.2$.

B. Results

In the first experiment, we evaluate how the planner's performance depends on the number of classes and objects involved. We query the sorting algorithm to sort 20, 25, 30, 35 and 40 randomly placed cubes assigned to 2, 3 and 4 classes. Due to different computational costs, we performed 100 trials for sorting 20 and 25 cubes, and 40 trials on sorting 30, 35 and 40 cubes. The results are shown in Fig. 5a and Fig. 5b. For the majority of the test cases we observe success rates of more than 75%, approaching 100% for fewer objects

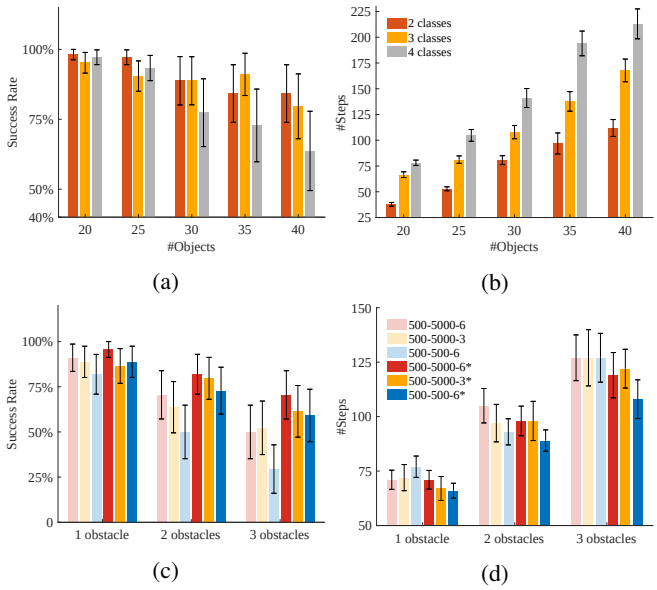


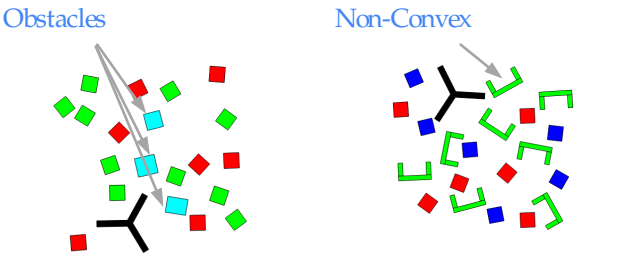
Fig. 5: (a) and (b) show success rates and number of steps respectively on sorting cubes with different number of classes. (c) and (d) show success rates and number of steps respectively for sorting 20 cubes in the presence of immovable obstacles. The success rate plots show 95% Wilson confidence intervals for planning success probability. The step plots show the average number of steps with standard error.

and classes. As the number of classes or objects increases, the success rate declines. We observe the same trend in the number of steps (actions), confirming that the difficulty increases as the number of objects and classes increases.

In our second experiment, we query the algorithm to sort scenes containing cubes and non-convex U-shaped objects, see Fig. 6. The U-shaped objects can easily entangle, or trap a cube, which makes it harder to rearrange these objects. The results are shown in Fig. 6. We observe that the algorithm can handle this case well, and the success rate is not significantly influenced. We observe, however, that the number of steps grows as the ratio of convex objects increases. This is due to the fact that the robot needs to spend additional actions on disentangling objects from each other.

In our third experiment, we query the algorithm to sort 20 cubes of 2 classes in the presence of 1, 2 and 3 immovable obstacles, see Fig. 6. These are the most difficult problems, as mistakenly pushing an object too close to an obstacle makes it very difficult to recover. In addition, the robot's motion is more constrained and needs to circumnavigate the obstacles. In this experiment, we also compare the learned rollout policy with the random one. For this, we run the planner with different parameter settings for the number of iterations $n_{\min}$, $n_{\max}$ and rollout depth $d_{\max}$ ($(500, 5000, 6)$, $(500, 5000, 3)$ and $(500, 500, 6)$ respectively).

We ran 40 tests in each test case and the results are shown in Fig. 5c and Fig. 5d. Recall that the learned policy is trained from sorting cubes in scenes with only one immovable obstacle. We observe that the learned policy successfully generalizes to problems with more than 1 obstacle, and manages to achieve higher success rates, where the performance of the random policy degrades. For both policies, a larger rollout depth and more iterations are unsurprisingly benificial. For the learned policy, however, we can even achieve good results with only 500 iterations, where the random policy performs much worse.



Non-Convex Ratio	10%	20%	30%	40%	50%
Success Rate	95.6% ±4.4%	93.3% ±6.2%	91.1% ±7.6%	93.3% ±6.2%	93.3% ±6.2%
#Steps	68.5 ±4.1	73.3 ±4.9	83.1 ±5.3	85.6 ±5.2	91.7 ±5.4

Fig. 6: Top: Example scenes with obstacle and non-convex objects. Bottom: Results of sorting 20 objects with different ratios of non-convex objects. The success rate numbers show the 95% Wilson confidence intervals of the planner's success probability. The step numbers are the mean step number with standard error.

VII. Conclusion

We addressed a planar non-prehensile sorting task, where a robot needs to separate many objects according to a user-defined class membership. In this problem, the robot needs to disentangle, circumnavigate and simultaneously push multiple objects. For this, task we adopted Monte Carlo tree search, and observed its effectiveness in reliably sorting up to 40 convex objects in 3 classes, despite only being equipped with a random rollout policy. Further, we observed the algorithm to be capable of sorting non-convex objects, and objects in the presence of obstacles, if equipped with a learned rollout policy.

These results are encouraging to further develop the use of Monte Carlo tree search for non-prehensile rearrangement. In this work, we did not emphasize the need to minimize the number of actions needed to achieve a sorted state. In future work, we intend to extend our approach in this direction. Further, while our current implementation achieves planning times of just a few seconds per action, we believe the efficiency can yet be greatly improved. Rather than replanning after every action from scratch, reusing previously grown search trees is a promising approach to save computation time. The challenge here, however, lies in coping with the prediction error made by the pushing model. Lastly, one limitation of MCTS is the restriction to discrete action spaces. This limits the use of this algorithm to rearrangement tasks like ours, where high-precision positioning of objects is not required.

REFERENCES

[1] G. Wilfong, "Motion planning in the presence of movable obstacles," *Annals of Mathematics and Artificial Intelligence*, vol. 3, pp. 131–150, March 1991.

[2] M. Stilman and J. J. Kuffner, "Navigation among movable obstacles: real-time reasoning in complex environments," *International Journal of Humanoid Robotics*, vol. 02, pp. 479–503, December 2005.

[3] M. Stilman and J. Kuffner, "Planning among movable obstacles with artificial constraints," in *International Journal of Robotics Research*, vol. 27, pp. 1295–1307, SAGE PublicationsSage UK: London, England, nov 2008.

[4] J. van den Berg, M. Stilman, J. Kuffner, M. Lin, and D. Manocha, "Path planning among movable obstacles: A probabilistically complete approach," *Algorithmic Foundation of Robotics VIII: Selected Contributions of the Eight International Workshop on the Algorithmic Foundations of Robotics*, pp. 599–614, 2010.

[5] D. Nieuwenhuisen, A. F. Van Der Stappen, and M. H. Overmars, "An effective framework for path planning amidst movable obstacles," *Algorithmic Foundation of Robotics VII: Selected Contributions of the Seventh International Workshop on the Algorithmic Foundations of Robotics*, vol. 47, pp. 87–102, 2008.

[6] N. Kitaev, I. Mordatch, S. Patil, and P. Abbeel, "Physics-based trajectory optimization for grasping in cluttered environments," *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 3102–3109, May 2015.

[7] W. C. Agboh and M. R. Dogar, "Real-time online re-planning for grasping under clutter and uncertainty," *Proc. IEEE-RAS Int. Conf. Humanoid Robots*, 2018.

[8] Muhayyuddin, M. Moll, L. Kavraki, and J. Rosell, "Randomized physics-based motion planning for grasping in cluttered and uncertain environments," *IEEE Robotics and Automation Letters*, vol. 3, pp. 712–719, April 2018.

[9] M. Stilman, J. U. Schamburek, J. Kuffner, and T. Asfour, "Manipulation planning among movable obstacles," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3327–3332, April 2007.

[10] O. Ben-Shahar and E. Rivlin, "Practical pushing planning for rearrangement tasks," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 4, pp. 549–565, 1998.

[11] A. Krontiris and K. Bekris, "Dealing with Difficult Instances of Object Rearrangement," *Proceedings - Robotics: Science and Systems*, July 2015.

[12] A. Krontiris and K. E. Bekris, "Efficiently solving general rearrangement tasks: A fast extension primitive for an incremental sampling-based planner," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3924–3931, May 2016.

[13] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "FFRob: Leveraging symbolic planning for efficient task and motion planning," *International Journal of Robotics Research*, vol. 37, no. 1, pp. 104–136, 2018.

[14] S. Han, N. Stiffler, A. Krontiris, K. Bekris, and J. Yu, "High-quality tabletop rearrangement with overhand grasps: Hardness results and fast methods," *Robotics: Science and Systems*, 2017.

[15] E. Huang, Z. Jia, and M. T. Mason, "Large-scale multi-object rearrangement," in *Proc. IEEE Int. Conf. Robotics and Automation*, IEEE, 2019.

[16] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[17] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[18] K. M. Lynch and M. T. Mason, "Stable pushing: Mechanics, controllability, and planning," *International Journal of Robotics Research*, vol. 15, pp. 533–556, December 1996.

[19] J. E. King, J. A. Haustein, S. S. Srinivasa, and T. Asfour, "Nonprehensile whole arm rearrangement planning on physics manifolds," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2508–2515, June 2015.

[20] J. E. King, M. Cognetti, and S. S. Srinivasa, "Rearrangement planning using object-centric and robot-centric action spaces," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3940–3947, May 2016.

[21] J. E. King, V. Ranganeni, and S. S. Srinivasa, "Unobservable Monte Carlo planning for nonprehensile rearrangement tasks," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 4681–4688, May 2017.

[22] J. A. Haustein, J. King, S. S. Srinivasa, and T. Asfour, "Kinodynamic randomized rearrangement planning via dynamic transitions between statically stable states," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3075–3082, June 2015.

[23] J. A. Haustein, I. Arnekvist, J. Stork, K. Hang, and D. Kragic, "Learning manipulation states and actions for efficient non-prehensile rearrangement planning,"

[24] W. Bejjani, R. Papallas, M. Leonetti, and M. Dogar, "Planning with a receding horizon for manipulation in clutter using a learned value function," 2018.

[25] L. Pinto, A. Mandalika, B. Hou, and S. S. Srinivasa, "Sample-efficient learning of nonprehensile manipulation policies via physics-based informed state distributions," *CoRR*, vol. abs/1810.10654, 2018.

[26] S. Elliott, M. Valente, and M. Cakmak, "Making objects graspable in confined environments through push and pull manipulation with a tool," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4851–4858, May 2016.

[27] M. Laskey, J. Lee, C. Chuck, D. Gealy, W. Hsieh, F. T. Pokorny, A. D. Dragan, and K. Goldberg, "Robot grasping in clutter: Using a hierarchy of supervisors for learning from demonstrations," in *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 827–834, Aug 2016.

[28] L. Chang, J. R. Smith, and D. Fox, "Interactive singulation of objects from a pile," in *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 3875–3882, May 2012.

[29] T. Hermans, J. M. Rehg, and A. Bobick, "Guided pushing for object singulation," in *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 4783–4790, Oct 2012.

[30] E. A, N. Hauff, and W. Burgard, "Learning to singulate objects using a push proposal network," 2017.

[31] M. Danielczuk, J. Mahler, C. Correa, and K. Goldberg, "Linear push policies to increase grasp access for robot bin picking," pp. 1249–1256, 08 2018.

[32] S. Zagoruyko, Y. Labbé, I. Kalevatykh, I. Laptev, J. Carpentier, M. Aubry, and J. Sivic, "Monte-carlo tree search for efficient visually guided rearrangement planning," *ArXiv*, vol. abs/1904.10348, 2019.

[33] J. E. King, V. Ranganeni, and S. S. Srinivasa, "Unobservable monte carlo planning for nonprehensile rearrangement tasks," in *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 4681–4688, IEEE, 2017.

[34] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.