Supplementary Material for
Compositional Diffusion-Based Continuous Constraint Solvers

Three videos are included in the supplementary directory:

1. `csp_solve_task1_triangle_packing.mp4` and `csp_solve_task4_robot_packing.mp4` are execution videos of solutions generated by our model Diffusion-CCSP.
2. `data_gen_task3_stability.mp4` shows the data generation simulation for task 3.

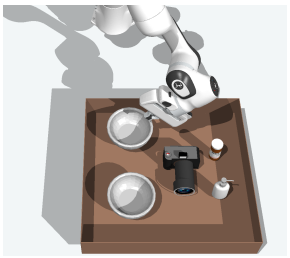
The appendix includes two sections. Appendix A discusses how to integrate Diffusion-CCSP for CCSP solving with task and motion planning (TAMP) algorithms. Appendix B details how our datasets are collected for each task and how the inputs are encoded for diffusion models. Appendix C includes further experiments on the number of samples it take Diffusion-CCSP to solve problems in each task.

A Integration with Task and Motion Planning Algorithms

So far, we have presented a generic solution to solving constraint satisfaction problems that involve geometric and physical constraints. However, assuming that the constraint graph is given as input to the algorithm. This approach can be directly used to solve particular tasks such as the pose prediction task in object rearrangements. Next, we illustrate how the proposed method can be integrated with a search algorithm to solve general task and motion planning (TAMP) problems, where the constraint graphs are automatically constructed based on the sequence of actions that has been applied and the goal specification of the task. For brevity, we will present a simplified formulation of TAMP problems. For more details, please refer to the recent survey [35].

Formally, given a space $\mathcal{S}$ of world states, a problem is a tuple $\langle \mathcal{S}, s_0, \mathcal{G}, \mathcal{A}, \mathcal{T} \rangle$, where $s_0 \in \mathcal{S}$ is the initial state (e.g., the geometry and poses of all objects), $\mathcal{G} \subseteq \mathcal{S}$ is a goal specification (e.g., as a logical expression that can be evaluated based on a state: $\mathit{in}(\mathrm{A}, \mathrm{Box})$ *and* $\mathit{in}(\mathrm{B}, \mathrm{Box})$), $\mathcal{A}$ is a set of continuously parameterized actions that the agent can execute (e.g., pick-and-place), and $\mathcal{T}$ is a partial environmental transition model $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Each action a is parameterized by two functions: precondition pre_a and effect eff_a. The semantics of this parameterization is that: $\forall s \in \mathcal{S}. \forall a \in \mathcal{A}. \mathit{pre}_a(s) \Rightarrow (\mathcal{T}(s, a) = \mathit{eff}_a(s))$.

The goal of task and motion planning is to output a sequence of actions $\bar{a}$ so that the terminal state s_T induced by applying a_i sequentially following $\mathcal{T}$ satisfies $s_T \in \mathcal{G}$. The state space and action space are usually represented as STRIPS-like representations: the state is a collection of state variables and each action is parameterized by a set of arguments. Figure 6 shows a simplified definition of the pick-and-place action in a table-top manipulation domain.



```
;; pick up x and and place x to p.
action pick-place(x, g, p, t)
  pre: valid-grasp(x, pose[x], g)        ;; g is the grasp pose on x
       valid-traj(x, pose[x], g, p, t)   ;; p is the target pose of x
       forall z. cfree(x, z, pose[x], t) ;; t is the robot trajectory
  eff: pose[x] := p                      ;; object x will be moved

goal: in(A, C, pose[A], pose[C]) and in(B, C, pose[B], pose[C])
```

The precondition-effect definition of a continuously parameterized action pick-place: pick and place object x.

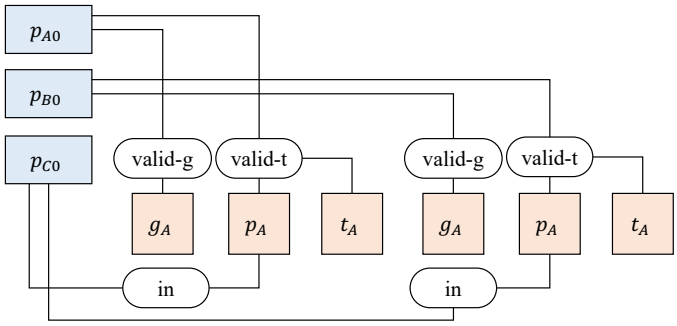
Figure 6: **Illustration of a simple task and motion planning problem.** The domain contains only one action: pick-and-place of objects. It contains three preconditions: g should be a valid grasp pose on object x, t should be a valid robot trajectory that moves x from its current pose to the target pose p, and during the movement, the robot and the object x should not collide with any other objects y. If successful, the object will be moved to the new location. The goal of the task is to pack both objects into the target box without any collisions.

```
pick-place(A, gA, pA, tA)
 valid-grasp(A, pA0, gA)
 valid-traj(A, pA0, gA, pA, tA)
 cfree(A, B, pB0, tA)
 cfree(A, C, pC0, tA)

pick-place(B, gB, pB, tB)
 valid-grasp(B, pB0, gB)
 valid-traj(B, pB0, gB, pB, tB)
 cfree(B, A, pA, tB)
 cfree(B, C, pC0, tB)

Goal
 in(A, C, pA, pC0)
 in(B, C, pB, pC0)
```

(a) A partially specified plan skeleton and its corresponding constraints.



(b) The corresponding TAMP constraint graph derived from the plan skeleton. cfree constraints and geometric features are omitted for brevity.

Figure 7: Example of a partially specified plan skeleton in task and motion planning, together with the corresponding set of constraints. p_{A0}, p_{B0}, and p_{C0} corresponds to the initial pose of objects.

A characteristic feature of TAMP problems is that the decision variables (i.e., the arguments of actions) include both discrete variables (e.g., object names) and continuous variables (e.g., poses and trajectories). Therefore, one commonly used solution is to do a bi-level search [36, 4]: the algorithm first finds a plan that involves only discrete objects (called a *plan skeleton*) and uses a subsequent CSP solver to find assignments of continuous variables, backtracking to try a different high-level plan if the CCSP is found to be infeasible. For example, consider the discrete task plan (also called *plan skeleton*): *pick-and-place*(A), *pick-and-place*(B). There are six parameters to decide: the grasps on A and B, the place locations of A and B, and the robot trajectory while transporting A and B. Based on the preconditions of actions and the goal condition of the task, we can obtain the constraint graph. Therefore, each solution to the CCSP constructed based on the plan skeleton corresponds to a concrete plan of the original planning problem. By combining the high-level search of plan skeletons and our constraint solver Diffusion-CCSP, the integrated algorithm is capable of solving a wide range of task and motion planning problems that involves geometric, physical, and qualitative constraints.

B Problem Domains and Data Generation

Next, we describe how data are generated and how variables are encoded for each task. All datasets are balanced, i.e. the number the examples involving different number of objects are the same. In neural network input encoding, all object dimensions and poses are normalized with regard to that of the container or shelf region.

B.1 2D Triangle Packing

Figure 8: Example collision-free configurations of triangles

Data. The data is generated by randomly sampling points in the square and, connecting them with each other and the edge points using the Bowyer–Watson algorithm, then shrinking each triangle a little to make space among them.

Encoding. We define the resting pose of a triangle as the pose when the vertex A facing the shortest side BC is at origin and its longest side AB is aligned horizontally. Its geometry is encoded three numbers, the length of the longest side $|AB|$ and the vector $(x, y) = \overrightarrow{AC}$. Its pose is encoded as the 2D coordinates of vertex A, along with the *sin* and *cos* values of the rotation θ from the testing pose.

B.2 2D Shape Arrangement with Qualitative Constraints

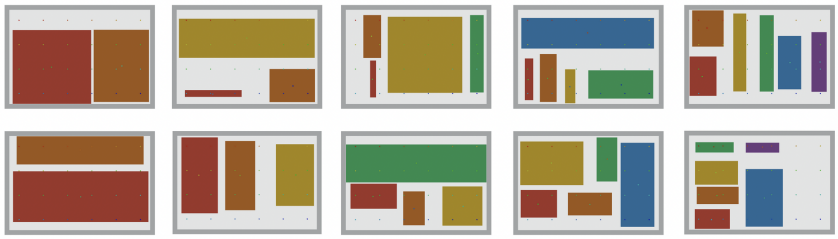


Figure 9: Example rearrangements of rectangles with `in`, `cfree`, and 11 types of qualitative constraints.

Data. The data is generated by recursively splitting the tray at different proportions until depth 3. For each resulting region, a random padding is added to each side. Regions whose area or side is too small are discarded. Labels of qualitative constraints are created by hand-crafted rules, e.g. `'close-to'` means the distance between two objects is smaller than the maximum width of two objects. All qualitative constraints include `'center-in'`, `'left-in'`, `'right-in'`, `'top-in'`, `'bottom-in'`, `'left-of'`, `'top-of'`, `'close-to'`, `'away-from'`, `'h-aligned'`, `'v-aligned'`.

Encoding. The resting pose of a rectangle is when its longer side is oriented horizontally. Its geometry is encoded using its width and length at resting pose. It's pose is the 2D pose of the centroid, along with the sin and cos encoding of the object's yaw rotation. Many problems require some objects to be at vertical positions in order for all the constraints to be satisfied.

B.3 3D Object Stacking with Stability Constraints

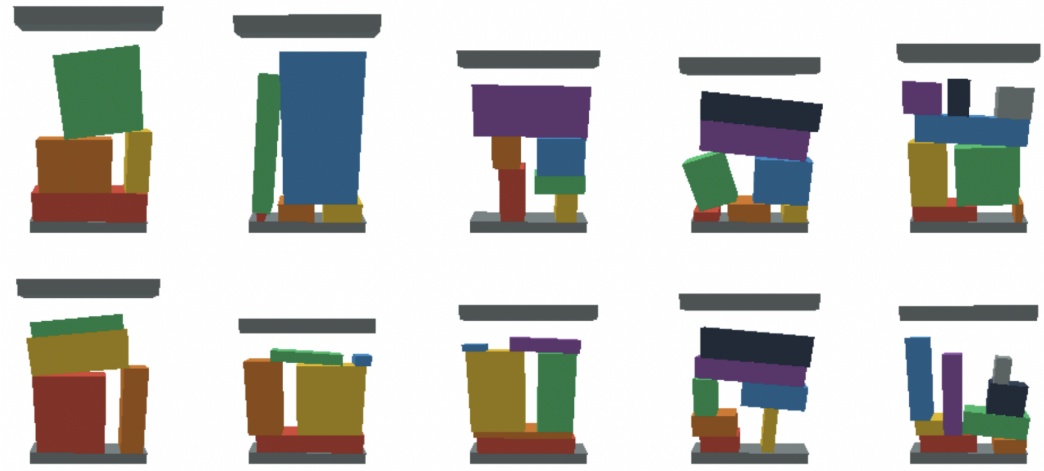


Figure 10: Example stable configurations of rectangles, with `in`, `cfree`, and `supported-by` constraints. There is at least one object that's supported by multiple objects.

Data. The data is generated by randomly splitting a 2D vertical region and shrinking and cutting each box. The resulting small boxes are initiated in PyBullet simulator and letting it drop until rest. We filter for configurations where the final state is stable. Then all objects are removed one by one to test that each intermediate configuration is also stable. An upper shelf is added to be just enough to fit the current configuration.

Encoding. The resting pose of a box is rotated so that the longer side is oriented horizontally. The rectangle's geometry is encoded using its width and length. The rectangle's pose is the 2D pose of the centroid, along with the sin and cos encoding of the object's $roll$ rotation.

B.4 3D Robot Packing with Robots

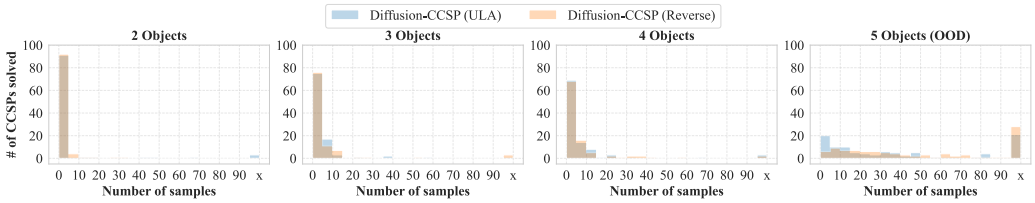
Figure 11: Example configuration of 3D shapes that enables the robot place each object in a given sequence without colliding into other objects already placed. All the bottles, dispensers, and bowls affords only side grasps.

Data. A fixed number of grasps are generated for each object that points directly to one of its five faces, e.g. $+x, -x, +y, -y, +z$. The data is generated by randomly splitting the tray into rectangle regions, then fitting into each region an object with random scale and orientation. The collision-free configuration is then tested by a TAMP planner to see whether a placement order and corresponding grasps can be found (by enumerating all combinations of order and grasps) so that the gripper at each grasp pose won't collide with any objects already in the goal region.

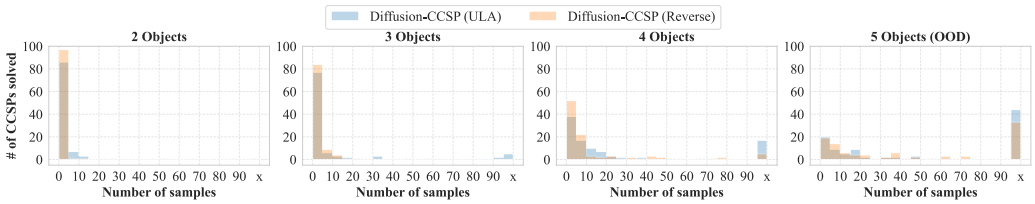
Encoding. The objects' geometry is encoded using axis-aligned bounding box (e.g., width, length, height). An object's grasp is encoded using a five-dimensional vector, indicating the face that the gripper points to. For example, $[0, 0, 0, 0, 1]$ is a top grasp. The object's pose is encoded using its 3D coordinate in the tray frame, along with the sin and cos encoding of the object's yaw rotation.

C Additional Sampler Comparison

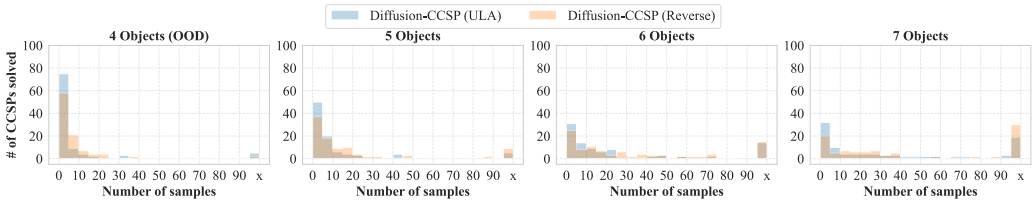
In the main experiment section, we let Diffusion-CCSP and baselines generate 10 samples for each CCSP and check if all constraints are satisfied. Here we let Diffusion-CCSP generate 100 samples on 100 problems, and plot the number of problems it took Diffusion-CCSP to solve for each task. In a batch of 100 CCSPs, it takes on average 0.01-0.05 sec for Diffusion-CCSP (Reverse) to solve each CCSP while 0.18-0.28 for Diffusion-CCSP (ULA) to solve each CCSP.



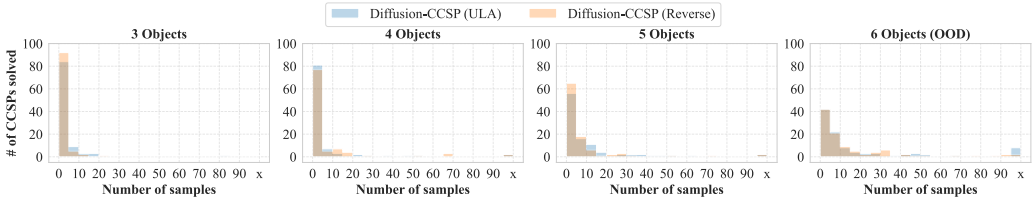
(a) Task 1: 2D Triangle Packing



(b) Task 2: 2D Shape Arrangement with Qualitative Constraints



(c) Task 3: 3D Object Stacking with Stability Constraints



(d) Task 4: 3D Object Packing with Robots

Figure 12: Number of Diffusion-CCSP runs it takes to solve 100 CCSPs. OOD means out-of-distribution problems that include more objects than trained on. x in x ticks means problems not solved within 100 runs.