Hybrid Discrete-Continuous Path Planning for Lattice Traversal

Santiago Franco[*1], Julius Sustarevas[*2], Sara Bernardini[1],

***Abstract*— Lattice structures allow robotic systems to operate in complex and hazardous environments, e.g. construction, mining and nuclear plants, reliably and effectively. However, current navigation systems for these structures are neither realistic, as they assume simplistic motion primitives and obstacle-free workspaces, nor efficient as they rely solely on global discrete search in an attempt to leverage the modularity of lattices. This paper tackles this gap and studies how robots can navigate lattice structures efficiently. We present a realistic application environment where robots have to avoid obstacles and the structure itself to reach target locations. Our solution couples discrete optimal search, using a domain-dependent heuristic, and sampling-based motion planning to find feasible trajectories in the discrete search space and in the continuous joint space at the same time. We provide two search graph formulations and a path planning approach. Simulation experiments, based on structures and robots created for the Innovate UK Connect-R project, examine scalability to large grid spaces while maintaining performances close to optimal.**

I. Introduction

Weight efficiency, high stiffness, and strength of lattice-like cellular structures have made them the focus of an increasing body of research in the fields of self-assembly and autonomous construction [1]–[4]. As illustrated in Fig. 1, often, in these applications, the system design includes a robot that needs to traverse a lattice to undertake assembly or inspection tasks. To do that, the robot must find a feasible path from one part of the structure to another, giving rise to a unique path planning problem. While the modular nature of the structure lends itself to discrete search, motion planning is carried out in a continuous joint space. In this paper, we propose a novel methodology that brings these two aspects together, satisfying the main assumptions of discrete search and leveraging the kinematic capabilities of modern robotics.

A significant work that explores robotic assembly of lattice structures is the project TERMES [5]. Inspired by termites, it features a team of small robots building a rectilinear structure from modular magnetic blocks. The robots can pick up the blocks, climb an existing structure and attach new blocks according to a building plan. The research explores assembly sequencing while maintaining structure traversability at any point during construction. A similar project by Jenett et al. [6], [7] propose a Material-Robot system that leverages a complex multi Degree of Freedom (DoF) robot assembling cub-octahedral and octahedron lattice elements. Here, multiple bipedal robots traverse and construct the structure simultaneously. However, both of these works adopt a simplified primitives-based approach for robot motion. In the case of

* Both authors contributed equally to this work.

[1] {Santiago.FrancoAixela},{Sara.Bernardini}@rhul.ac.uk, Royal Holloway University of London. [2] info@ross-robotics.co.uk, Ross Robotics Ltd

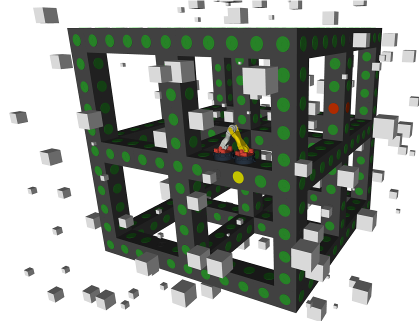


Fig. 1: Lattice Structure Traversal Problem: Lattice components are shown in grey, obstacles in white and connectors (sockets) in green. Robot starting connector marked in yellow, end connector in red.

TERMES, the robot relies on mechanical solutions to assure motion success and can only move orthogonally, climbing up or down. The Material-Robot system uses only a handful of human prescribed and pre-computed primitive motions. While use of primitives simplifies the path finding problem and allows the use of standard discrete search algorithms such as A^*, it fails to leverage the capabilities of multi-DoF robot kinematics and requires strong assumptions such as an obstacle free environment and a completely rigid lattice structure. These assumptions compromise the applicability of self-assembly structures to real-world applications.

This paper is inspired by recent work on self-assembling structures for nuclear decommissioning [8]. In this setting, information about the environment and structure is available, but obstacles might be present. Furthermore, as exposure to radiation is undesirable, minimizing the time the robot takes to traverse the structure becomes crucial. The lattices in this paper are based on the structural robot prototype discussed in [8]. Therefore, we propose a methodology on how joint space robot motion planning and discrete lattice search can be coupled, allowing complex collision-aware motions. The contributions of this paper are as follows: 1) we propose two graph formulations for path finding in lattice structures, requiring different computational efforts; 2) we propose a discrete planning approach for these graphs; and 3) we carry out an extensive evaluation of our approach in simulation.

The structure of this paper is as follows. Section II reviews literature on self assembly systems, climbing robots and discrete search. In Sec. III, we describe how the lattice, the robot and the planning problem is modelled. We then propose graph formation in Section IV and search approaches in Section V. Section VI offers an evaluation of our techniques in simulation and discusses their qualitative and quantitative properties. Conclusions are presented in Section VII.

II. Related Work

1) Climbing Robots and Lattice Traversal: Together with self-assembly and construction, systems similar to ours have been used for inspection [9]–[11] and maintenance [12]. A common type of robot considered for these problems is the *Inchworm* archetype [13]. These robots consist of a single kinematic chain that is symmetrical about a revolute joint. The two ends of the kinematic chain are used to attach the robot to the environment creating a biped-like motion.

Considering structure traversal, leveraging structure discreteness could help find paths over it using established search methodology. The Material-Robot system [6], for example, after defining motion primitives, uses A^* with the Manhattan heuristic. However, further work by Jenett et al. [14] examines compliance and deformation within the lattice structures and thus highlights the necessity to plan online.

When motion primitives are not assumed, finding optimal intermediate states or gait sequences for the inchworm robot becomes a challenge. A work by Yang et al. [15] describes how a graph can be constructed by considering the end-effector landing locations as the base for future steps. To find a feasible path, they use *Breadth First Search* and validate each edge via an online call to motion planning. Determining these gaits or attachment locations is also explored by Zhu et al. [16], who focus on biped robots climbing over truss-like structures. The robot's kinematic capabilities determine its ability to perform a collision free transition from one truss to another, in turn informing global path planning. The search space is small as only a few trusses are considered, however determining the grip locations as part of the problem increases its complexity. The related problem of climbing tree branches is studied by Lam et al. [17], [18]. The robot used in these works has a flexible continuum body, but locomotion-wise is similar to the inchworm archetype. These works shows how tree branches can be described as a graph and locations along the branches are discretized to allow for dynamic programming search. However, the problem is simplified by preventing the robot switching between branches, which reduces the graph's size.

Pagano et al. [19] proposes a more complex scenario where incomplete information about the world is assumed. In this work, an inchworm robot uses depth sensors to allow an iterative *Line of Sight Tree* approach to find intermediate robot states. Meanwhile, a potential field motion planning is used to find the joint space motion that the robot needs to follow. While this approach runs online and is highly flexible, it is far from optimal as it requires the robot to backtrack frequently to gather additional information.

2) Discrete Search: The robotics works discussed above tend to split the discrete and continuous parts of the problem and apply different methods to each part. Divide and conquer approaches are also seen in discrete search literature, e.g. the TIM planner [20], where a generic planner uses a specialized planner to solve a subset of the problem. Semantic attachments follow a similar strategy [21], [22].

In this paper, we run A* multiple times as we learn the real cost of actions. LPA* [23] is an incremental type of A* that repeatedly calculates an plan from a fixed initial state to the goal state, when some of the action costs change. The first search is identical to A*, following searches place nodes in a priority queue for reevaluation if a node becomes inconsistent due to changes to the predecessor edge's costs. Expansions are ordered by f-value and tie-breaked by $min(g, rhs)$, where rhs is a look-ahead value based on the minimum g values of the node's predecessors. LPA* is guaranteed to expand every state at most twice per search episode. LPA* is not guaranteed to be correct (always find optimal plan) when using admissible but inconsistent heuristics, unlike A*. In our application, the A* search effort is insignificant for most problems, however for the largest discrete search spaces LPA* should be considered.

D* [24] can be used to generate optimal paths in a partially known environment. In our case, the discrete environment is fully known, but edges need to be validated because obstacles or deformations require *ad hoc* solutions. We ignore obstacles in the discrete search space because validating all expanded edge costs would be too expensive.

III. Problem Modelling

In this section, we outline how the lattice structure and the robot are modeled and state the path planning problem we tackle. We also describe the software framework we used.

A. The Lattice Structure

To capture the construction and self-assembly use cases presented in the literature [6], [8], we generalize the lattice structure as a 3-dimensional grid of homogeneous cubes of side d ($d = 0.35\,\text{m}$ in our experiments). Fig. 1 shows how cubes form an overall lattice with obstacle cubes in white. Each exposed face of each lattice cube has a *connection socket* that a robot can attach itself onto. Both lattice and obstacle cubes have an associated collision box, but there is a gap (1 cm) between the socket frame and the collision box to allow easy motion planning when attaching to the socket.

A socket coordinate can be represented as a vector of integers $s = (x, y, z, f)$, where $x, y, z \in \mathbb{Z}, f \in \{0, 1, 2, 3, 4, 5\}$. The x,y,z components translate into multiples of cube widths. The face value f denotes the index of cube faces expressed as the rotations $\{\mathbf{I}, \mathbf{R_y}(\frac{\pi}{2}), \mathbf{R_x}(-\frac{\pi}{2}), \mathbf{R_y}(-\frac{\pi}{2}), \mathbf{R_x}(\frac{\pi}{2}), \mathbf{R_y}(\pi)\}$, where $\mathbf{I}$ is the identity matrix and $\mathbf{R_x}, \mathbf{R_y}$ are rotation matrices about the x and y axes. Fig. 2 illustrates the socket naming and frame orientation conventions.

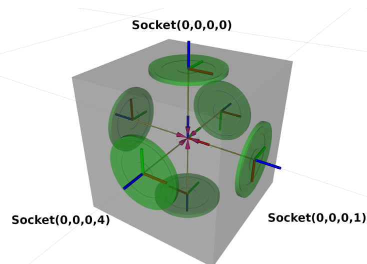


Fig. 2: Socket frame x,y,z axes shown in red, green and blue.

The conventions above allow a compact description of socket coordinate in grid-world. The homogeneous transfor-

mation matrix denoting the socket frame can be found as the following multiplication of translations and rotations:

$$T_s = Trans(d \cdot (x,y,z))Rot(\mathbf{R}_f)Trans(\frac{d}{2} \cdot (0,0,1)) \quad (1)$$

B. The Multi-Task Bot

Following literature, we adopt the inchworm robot archetype. For our experiments we simulated the inchworm type robot created for the Connect-R project [25]. We refer to the robot as Multi-Task Bot (MTB). We model the MTB as a symmetrically arranged 5-DOF arm. The robot is illustrated in Fig. 3. In our experiments, we use a robot with link lengths of $d_1 = 11.015$ cm, $d_2 = 5.065$ cm, $d_3 = 33.6$ cm. Robot joints 2-4 rotate about their y axes, while joints 1 and 5 rotate about z axis. Rotation limits for joints 1-5 are $\pm180°$, $\pm95°$, $\pm121°$, $\pm95°$,$\pm180°$, respectively. We also assume all joints have the same velocity limit, set to $\pm0.5\,\mathrm{rad\,s^{-1}}$ and gear reduction ratio of 90.

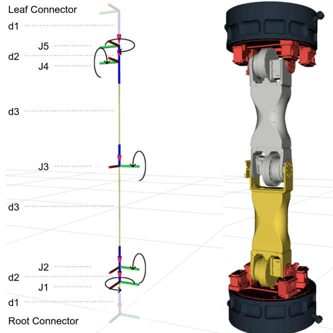


Fig. 3: Connector frames and joints of the MTB robot. Joints 1 and 5 rotate about the z-axis, while joints 2-4 rotate about the y-axis.

The robot is designed to be symmetrical about joint 3, making the ends of the kinematic chain interchangeable. This also means that we only require one kinematic model to describe the robot both ways along the chain. For the robot to fix itself to the structure, a connector frame must fully overlap a lattice socket frame, orientation included. As a convention, we say *root connector* to mean the one that remains fixed during robot motion. Given a homogeneous transform to a root connector T_R and a joint state $\mathbf{j} = (j_1, j_2, j_3, j_4, j_5)$, the leaf connector frame can be found via Forward Kinematics $T_L = FK(T_R, \mathbf{j})$. To model the reverse, we define a robot joint *inverse* function as follows:

$$\begin{aligned} inverse(\mathbf{j}) = (-j_5, -j_4, j_3, -j_2, -j_1), then \\ if\ T_L = FK(T_R, \mathbf{j}), it\ follows: \\ T_R = FK(T_L, inverse(\mathbf{j})) \end{aligned} \quad (2)$$

C. Software Framework

The lattice and robot described in this section are modeled using tools from the Robot Operating System (ROS) ecosystem. All transformation calculations are carried out using the ROS tf and PyKDL frameworks. Robot motion planning uses the MoveIt! [26] software package, where the lattice structure populates the planning scene of the robot. The robot inverse kinematics are computed via the Trac-ik solver [27]. We use two joint-space planners, RRTConnect [28] and RRTStar [29], as they are implemented by the Open Motion Planning Library (OMPL) [30]. Both planners are sampling-based and adopt the Rapidly exploring Random Tree (RRT) methodology. The RRTStar planner is asymptotically optimal and we set its optimization objective as to minimize the maximum element of joint distance traveled.

D. Path Planning Problem Statement

Consider the problem in Fig. 1. A multi-task robot is attached to the lattice structure at a starting socket s_0 and its joint configuration is known. The position and size of the lattice cubes, the obstacles and the set $\mathbb{S}$ of all free sockets on the lattice are also assumed to be known. The robot is tasked with connecting to any free socket on a goal cube $g = (g_x, g_y, g_z)$ located on the lattice, i.e. the set of goal sockets is $S_g = \{(g_x, g_y, g_z, 0...5)\} \subset \mathbb{S}$. The planner must find a sequence of sockets s_i and corresponding joint space trajectories $J_{i,i+1}$ taking the robot from s_0 to one of the goal state $s_g \in S_{goal}$, such that each $J_{i,i+1}$ is feasible and collision free.

IV. Graph Formation

In this section we tackle the lattice structure path planning problem. We first examine the problem statement presented in the previous section and propose two alternatives of how lattice and robot descriptions can lead to a discrete search graph construction. The vertices of these graphs will correspond to robot state, while edges - to robot motion. Then, in Section V, we show that calling continuous joint space planners to retrieve graph edge costs during discrete search leads to poor performance and scalability. To address this, we describe how computing an offline cache can provide asymptotically-minimal estimated edge costs and in doing so allow an iterative search-validation approach.

A. Single Socket ab-Graph

A naive way to form a search graph is by choosing on the *neutral* joint configuration that the MTB takes when attached to a single socket. We use the all-zero configuration $\mathbf{j} = \mathbf{0}$, corresponding to the MTB fully stretching out (see Figure 4a). In this way, tuple $v = (s, \mathbf{0})$ corresponds to placing the MTB root connector at T_s and setting all joints to zero. This tuple describes the MTB state completely and, in turn, defines a graph vertex v. An edge of such a graph corresponds to the MTB *hopping* from one socket to another, starting and ending with $\mathbf{j} = \mathbf{0}$, see Fig. 4. Only vertices with mutually reachable sockets share an edge, this is discussed in Sec. IV-C. Computing the joint space trajectory for an edge between sockets a and b is thus composed of four steps:

1) Find the inverse kinematics solution $\mathbf{j}^* = IK(T_a, T_b)$.
2) Compute the collision aware trajectory J_1 taking the MTB from state $(a, \mathbf{0})$ to $(a, \mathbf{j}^*)$.
3) Compute the collision aware trajectory J_2 taking the MTB from state $(b, inverse(\mathbf{j}^*))$ to $(b, \mathbf{0})$.
4) Concatenate the trajectories: $J_{a,b} = [J_1, J_2]$.

We call this graph formulation the Single Socket *ab*-graph, signifying that each edge corresponds to two sockets, while

vertices to one. Note that the joint *inverse* function allows us to use a single robot kinematics model during computation. For $J_{a,b}$ to be performed on real hardware, the J_2 component needs to be inverted back.

The *ab*-graph's main advantage is that each vertex in the graph can be described by using a single socket. However, two expensive joint-space motion plans must be formulated to compute the edge costs. Also, as the neutral state has the same configuration for all sockets no matter of their surroundings, there might be cases in which the MTB ends up colliding with nearby obstacles, invalidating the corresponding vertices.

B. Double Socket abc-Graph

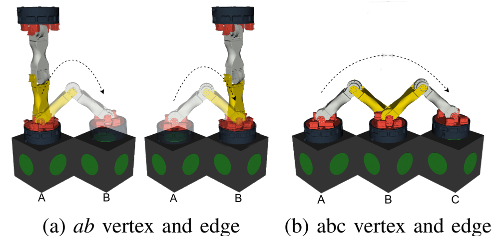
A graph can also be built by using two sockets for a vertex and three for an edge (see Figure 4b). We call this graph Double Socket *abc*-graph, signifying three sockets per edge where socket b is common to both the initial and final vertex. Here, a vertex is a tuple $v = (a, b, \mathbf{j}_{a,b})$, s.t. $\mathbf{j}_{a,b} = IK(T_a, T_b)$. The pair of sockets a, b must be mutually reachable. Assuming that, for each socket $s \in \mathbb{S}$, we have access to the set of reachable sockets $\mathbb{C}_s \subset \mathbb{S}$, the pairs then are $(a, b), \forall a \in \mathbb{S}, \forall b \in \mathbb{C}_a$. We explain in the next section how the set $\mathbb{C}_s$ is obtained. Note that there can be multiple or even infinite (in case of high-DoF robots) inverse kinematics solutions between two sockets. For this reason, when we create the graph data structure, we find a single $\mathbf{j}_{a,b}$ of each vertex. A path planning algorithm operating on this graph will not consider alternative $\mathbf{j}_{a,b}$ and as such, solutions found or optimality properties are restricted to the specific set of $\mathbf{j}_{a,b}s$ associated with the vertices. Furthermore, the vertices $(a, b, \mathbf{j}_{a,b})$ and $(b, a, inverse(\mathbf{j}_{a,b}))$ correspond to the exact same robot state. To avoid unnecessary vertices, we store unique vertices only by using alphanumerical sorting.

The *abc*-graph dominates the *ab*-graph in terms of completeness. The *ab*-graph requires to fully extend the MTB after connecting to a new socket while the *abc*-graph only requires that a physical connection is possible between two sockets. Hence, there are multiple cases where a socket is reachable in the *abc*-graph while unreachable in the *ab*-graph. Any solution in the *ab*-graph has at least one equivalent solution in the *abc*-graph while the opposite is not necessarily true. This is critical when dealing with obstacles.

Any particular set of reachable sockets $\mathbb{C}_s$ can be obtained by appropriately translating and rotating sockets inside $\mathbb{C}_\mathbf{0}$. In our implementation, we compute the homogeneous transformations for all sockets in the CM, perform the frame transformation T_s and cast the transforms back into socket tuples. The resulting set is then intersected with the set of free sockets $\mathbb{S}$ on a lattice, obtaining $\mathbb{C}_s \subset \mathbb{S}$.

Finally, an edge of the *abc*-graph corresponds to a single *hopping* motion between any two reachable vertices that share a socket, with the robot start and end joint configurations being the ones associated with the vertices, see Fig. 4. The joint space trajectory for the motion can be computed as follows:

1) Find the matching socket b between (a, b) and (b, c) vertices.
2) Appropriately invert the joint configurations associated to (a, b) and (b, c) vertices to obtain $\mathbf{j}_{b,a}$ and $\mathbf{j}_{b,c}$.
3) Compute the collision aware trajectory $J_{(a,b),(b,c)}$ taking the robot from state $(b, \mathbf{j}_{b,a})$ to $(b, \mathbf{j}_{b,c})$



(a) *ab* vertex and edge (b) abc vertex and edge

Fig. 4: Robot motions for *ab* and *abc* graphs. Final vertex positions are shown in full opacity, intermediate configurations are transparent, and the edge motions are show by the arrows.

C. Connectivity Map

The above descriptions on how the *ab* and *abc* graphs can be constructed rely on knowing a set of reachable sockets $\mathbb{C}_s \subset \mathbb{S}$ to which the robot can connect while still being attached to socket $s \in \mathbb{S}$. To find these sockets, we compute the robot's Connectivity Map (CM) $\mathbb{C}_\mathbf{0}$ empirically. We create a fully populated grid world of size n centered at world origin. Grid size is chosen chose so it encapsulates robot reach volume. In our case, $n = 7$, i.e. all cubes between $(-3, -3, -3)$ to $(3, 3, 3)$ exist. All six faces of all cubes in the grid have sockets. Collision detection for this setup is disabled. We attach the robot to socket $(0, 0, 0, 0)$ and compute the inverse kinematics $IK(T_\mathbf{0}, T_{s \in \mathbb{S}_{grid}})$ to all sockets in this grid. The sockets for which an IK solution is found form the robot connectivity map $\mathbb{C}_\mathbf{0}$. In our case, the CM is comprised of 53 sockets.

Any particular set of reachable sockets $\mathbb{C}_s$ can be obtained by appropriately translating and rotating sockets inside $\mathbb{C}_\mathbf{0}$. In our implementation, we compute the homogeneous transformations for all sockets in the CM, perform the frame transformation T_s and cast the transforms back into socket tuples. The resulting set is then intersected with the set of free sockets $\mathbb{S}$ on a lattice, obtaining $\mathbb{C}_s \subset \mathbb{S}$.

D. Graph Validation

For both types of graphs, a collision check is performed so only valid vertices are considered. The Connectivity Map is used to determine the existence of the graph edges. For single socket *ab*-graphs, the edges exist for all vertices $v, w = (s, \mathbf{0})$ such that $w_s \in \mathbb{C}_{v_s}$. For double socket, *abc*-graphs, edges exist for all vertices $v, w = (a, b, \mathbf{j}_{a,b})$ such that w_a or w_b is in $\mathbb{C}_{v_a} \cup \mathbb{C}_{v_b}$. Note that this check also guarantees that v and w share a socket. Finally, we define the edge cost as the time taken by the collision free edge motion, which is dependent on the structure and obstacles. Thus, at the graph construction stage, edge costs are left unset.

V. Path Planning For Traversal of Lattices

In Section IV, we describe how the *ab* and *abc* graphs can be constructed to capture the lattice structure and a robot's

ability to traverse it. We can now define a global search problem that captures the path planning problem stated in Sec. III-D. Given the set of free sockets $\mathbb{S}$, the goal cube g that defines the goal sockets $\mathbb{S}_g$ and a starting socket s_0, the global search problem can be defined as the tuple $\Pi = \langle \mathbb{V}, v_0, \mathbb{V}_g, \mathbb{J} \rangle$, where $\mathbb{V}$ is the set of the graph vertices constructed from $\mathbb{S}$, v_0 is the initial vertex corresponding to s_0, $\mathbb{V}_g \subset \mathbb{V}$ is a set of goal vertices such that the robot is attached to any of the goal sockets in $\mathbb{S}_g$ and $\mathbb{J}$ represents all the available actions, i.e. the joint trajectories that a robot can perform. A plan $\pi = J_0, J_1, ..., J_n$ is made up of consecutive actions that take the robot from v_0 to a state $v_g \in \mathbb{V}_g$. A plan is optimal if there is no alternative plan reaching any $v_g \in \mathbb{V}_g$ with a smaller cost. The function $c(J)$ represents the cost of each action, that is the time of the joint trajectory J. The search problem seeks to minimize $c(\pi) = \sum_{J \in \pi} c(J)$. To distinguish between the two types of graphs, we use $v = (s, \mathbf{0}) \in \mathbb{V}_{ab}$ and $v = (a, b, \mathbf{j}_{a,b}) \in \mathbb{V}_{abc}$.

A. A^* with Online Motion Planning

To grasp an intuition on the search problem, we first examine how a naive A^* approach performs on both graphs. The graph connectivity is computed offline, but every time A^* attempts to obtain the edge cost of a new edge, the RRTConnect joint space planner is called to find a joint trajectory J taking the robot from one vertex state to the other as described in Sections IV-A and IV-B. The RRT-Connect planner is fast but not optimal and is used only for illustration. Using an optimal planner for this test would lead to very poor performance (we obtained no meaningful data in our experiments). Also, efficient A^* planning requires an informed admissible heuristic. Here, we use a *Weighted Euclidean heuristic*. We describe and prove the admissibility of this heuristic in an upcoming section. The results of testing A^* with online motion planning are shown in Fig. 5.

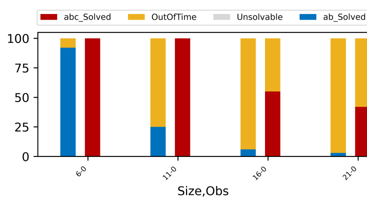


Fig. 5: Number of Solved and OutOfTime (2 h) instances over 100 trials with online validation. Lattice sizes range from 6 to 21.

Figure 5 shows that online edge cost validation scales up poorly, even without obstacles, as the lattice size increases. Section VI shows how a cache-based approach scales better. Note that using greedy search algorithms would generally reduce the number of node expansions and hence speed up finding solutions. However, we next present an iterative method which preserves optimality while greatly reducing average node expansion costs. If optimality is not required, the proposed iterative approach could easily be adapted to greedy search methods as well.

B. Iterative A^* With Cached Costs

Figure 5 shows that calculating a joint space solution for each node expansion is impractical even in the best case scenario (without any obstacles). Instead, we propose an iterative A^* with cached costs approach. Firstly, we compute an offline cache of asymptotically optimal costs associated with the *ab* and *abc* motions. This cache can then be used to obtain admissible and consistent cost estimates during A^* vertex expansion. We also use these costs to derive the weights for the *Weighted Euclidean heuristic*. To ensure the paths are feasible, A^* will be called iteratively, performing path validation between iterations.

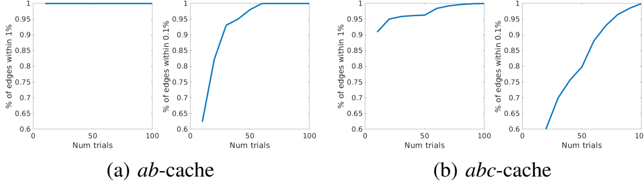
1) Costs Cache Computation: We compute the cache by performing multiple joint space plans for *ab* and *abc* motions using the RRTStar planner. We use the same grid setup as in Section IV-C, disable collisions with obstacles, but keep MTB self collisions. Vertex pairs considered for *ab* cache are shown in Eq. 3, for *abc* in Eq. 4:

$$((0,0,0,f),b) \forall f \in \{0..5\}, \forall b \in \mathbb{C}_\mathbf{0} \quad (3)$$

$$((0,0,0,f),a),((0,0,0,f),b) \forall f \in \{0..5\}, \forall a,b \in \mathbb{C}_\mathbf{0} \quad (4)$$

We do not assume a single IK solution for $((0,0,0,f),a)$ and $((0,0,0,f),b)$. IK is called every time, this way many IK solutions are tried and the approach is applicable to higher DoF robots. Furthermore, as described in Sec. III, cube sockets have specific orientations. Therefore, the whole cache data structure cannot be computed only for $f = 0$. For example, $\mathbf{I} \cdot \mathbf{R_y}(-\frac{\pi}{2}) \neq \mathbf{R_x}(\frac{\pi}{2}) \cdot \mathbf{R_y}(-\frac{\pi}{2})$. This does not affect the CM computation, but when planning a motion, joints could travel a different distance and so all sockets on cube $(0,0,0)$ are tested. Structure-dependent symmetries could be exploited to speed up cache computations. A total of 100 trials are computed for each edge, in order to capture all possible IK solutions, and the fastest is taken. The resulting cache thus stores the minimum costs found for any edge on v, v' on the $(0,0,0)$ cube $\mathbb{K}_\mathbf{0}(v,v') = \min(c(v,v'))$. $\mathbb{K}_\mathbf{0}$ is computed once for *ab* and *abc* motions and is specific to robot root connector similarly to CM. The cache can be applied to any edge on the lattice by expressing the x, y, z components of the vertex sockets relatively to the root connector socket. However, unlike CM, computing the $\mathbb{K}_\mathbf{0}$ is expensive. The *ab* cache involves 318 vertex combinations and takes about 25h using a consumer grade laptop. The *abc* cache involves 8268 vertex combinations and takes about 650h. To perform these computations we used the computing cluster described in Section VI. The cache is *asymptotically-minimal* because we are using the RRTStar planner, which is asymptotically optimal. Cache optimality is subject to number of trials and the RRTStar time out threshold (we use 5 s). Fig. 6 shows how during both types of cache computation the minimum costs over increasing number of trials converges to the min over all trials.

2) Heuristics: In addition to edge costs, the cache allows us to derive an informative *Weighted Euclidean heuristic* h_ε to be used with A^* search. A heuristic $h(v \in \mathbb{V})$ is an estimation of the minimum cost of a feasible plan π, taking



(a) *ab*-cache (b) *abc*-cache

Fig. 6: % of edges whose min *costs* over *num* trials are within an error margin of 1% and 0.1% of min over all trials.

Algorithm 1: Iterative A*

Data: $\Pi = \langle \mathbb{V}, v_0, \mathbb{V}_g, \mathbb{K} \rangle, RRTC$
Result: π

1. $ValActions \leftarrow \varnothing$; $c(A) \leftarrow EstimatedCosts()$; $SolutionFound \leftarrow \bot$;
2. **while** $\neg SolutionFound$ **do**
3. $\quad \pi, SolutionFound \leftarrow Astar(\Pi)$;
4. $\quad$ **if** $SolutionFound$ **then**
5. $\quad\quad validated = \top$;
6. $\quad\quad$ **for** $a \in \pi$ **do**
7. $\quad\quad\quad$ **if** $a \in ValActions$ **then**
8. $\quad\quad\quad\quad$ **continue**;
9. $\quad\quad\quad feasible, cost \leftarrow RRTConnect(\langle v, v' \rangle \in a)$;
10. $\quad\quad\quad$ **if** $feasible$ **then**
11. $\quad\quad\quad\quad c(a) \leftarrow cost$;
12. $\quad\quad\quad\quad ValActions \leftarrow a$;
13. $\quad\quad\quad$ **else**
14. $\quad\quad\quad\quad \mathbb{A} \leftarrow \mathbb{A} \setminus a$; $validated \leftarrow \bot$;
15. $\quad\quad\quad\quad$ **break**;
16. $\quad\quad$ **if** $validated$ **then**
17. $\quad\quad\quad$ **return** π
18. $\quad$ **return** $\varnothing$ `// No Feasible Solution found`

v to $\mathbb{V}_g$. A heuristic is admissible if $h(v) \leq h^*(v)$ where $h^*(v)$ is the costs of a true optimal plan. A heuristic is consistent if both $h(v) \equiv 0 | \forall v \in \mathbb{V}_g$ and $h(v) \leq c(v,v') + h(v') | \forall v \in \mathbb{V} \setminus \mathbb{V}_g$ where v' and v share an edge. Consistency implies admissibility. Both properties are desirable as admissibility guarantees optimality and consistency leads to fewer node expansions and shorter search times. First we define a distance function between two sockets s, s'. We then extend this definition to define h_ε for *ab* and *abc* graphs:

$$dist(s,s') = ||(s_x, s_y, s_z) - (s'_x, s'_y, s'_z)||_2 \tag{5}$$

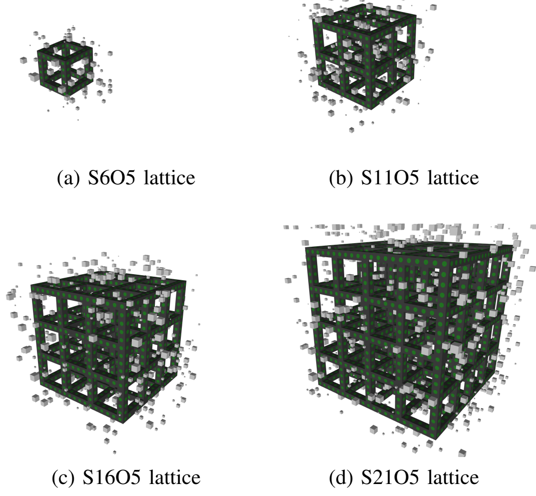
$$h_{\varepsilon,ab}(v \in \mathbb{V}_{ab}, g) = \omega \cdot dist(v_s, g) \tag{6}$$

$$h_{\varepsilon,abc}(v \in \mathbb{V}_{abc}, g) = \omega \cdot \min(dist(v_a, g), dist(v_b, g)) \tag{7}$$

The weight ω in Eqs. (6, 7) is determined using the cache $\mathbb{K}_\mathbf{0}$. For both graphs, it is the minimum (over all edges $([v,v'] \in \mathbb{K}_\mathbf{0})$ ratio between edge costs $\mathbb{K}_\mathbf{0}(v,v')$ and the maximum distance between sockets involved with the edge. That is, $s, s' \in \{v_s, v'_s\}$ for *ab* cache and $s, s' \in \{v_a, v_b, v'_a, v'_b\}$ for *abc*. In other words, Eq 8 describes capturing the minimal costs to travel the longest Euclidean distance.

$$\omega = \min_{[v,v'] \in \mathbb{K}_\mathbf{0}} \frac{\mathbb{K}_\mathbf{0}(v,v')}{\max_{s,s' \in \{v,v'\}} dist(s,s')} \tag{8}$$

Due to the triangle inequality, an unweighted Euclidian distance is a consistent heuristic if $costs(v,v') \geq d(v,v')$. Therefore, assuming the costs $c(v,v') = \mathbb{K}_\mathbf{0}(v,v')$ are minimal, we only need to ensure that $|h_\varepsilon(v') - h_\varepsilon(v)| \leq c(v,v')$ to keep consistency. The trivial solution would be to make $\omega \equiv 0$, but that would not be informative. Eq. (8) ensures that there is no action for which the change in heuristic values from v to v' is bigger than the cost of moving from v to v'. The same logic applies to both graphs because we defined the distance function for *abc* vertices as the pair-wise minimum distance between any sockets $((0,0,0,f), v_a, v_b) \in \{v, v'\}$. The admissibility and consistency of the heuristic using these weights is only guaranteed up to the correctness of the asymptotically-optimal cost estimates.



(a) S6O5 lattice (b) S11O5 lattice

(c) S16O5 lattice (d) S21O5 lattice

Fig. 7: Cuboid lattice structure used for testing. For example, S11O5 corresponds to lattice of side length 11 and 5% of bounding $17 \times 17 \times 17$ cube volume being obstructed by obstacles. A video of experiments is available here

3) Iterative A^*: To reduce the number of calls to the joint space planner, we perform iterative search and validation using the following steps: 1) The graph is populated with vertices and edges. Edge costs are estimated using $\mathbb{K}$. 2) An A^* search is performed on the graph. 3) We then validate all edges in the path found using the RRTConnect planner. During this step, the collision-free joint-trajectory J is stored with the edge and costs are updated with the duration of J. This also includes removing edges, when joint state planner finds no solution. 4) We repeat steps 2,3 using a persistent graph data structure. That is, the edges are validated or removed in every iteration. We are finished once we have found a feasible path such that all edges in the path are validated. Please see Algorithm 1 for details.

4) Optional Replanning Window: The first time A^* search is performed, the path is found using the estimated edge costs, let the costs of this path be $c(\pi^*)$. The algorithm 1 then goes on to validate the edges and call A^* again until the whole path is feasible, resulting in a valid plan π. However, π is not guaranteed to be the cheapest plan, because there might be alternative cheaper paths using yet to be validated edges. On the other hand, searching and validating edges takes time. Hence, optimally, one could keep calling Iterative A^* for the time duration $c(\pi) - c(\pi^*)$. This replanning window allows to only search for more optimal paths if a more optimal plan is possible when taking into account the replanning costs. In Section VI, we show that in the majority of experiments the first feasible plan returned by iterative A* is also the best

one once the replanning time effort is taken into account.

VI. EVALUATION

1) Experimental Setup: To evaluate our approach, we use four lattice-structures as presented in Fig. 7. The cuboid lattices range in size form 6 to 21 and are made up of repeating sub-lattices. In order to obstruct inside and outside the structure, we first define a bounding cube. We do this by adding 3 cube units along each direction, e.g. a bounding cube for lattice of size 6 would be size 12. Then, for each free cube-point we place an obstacle with a probability 0% to 50% in 5% increments, creating different obstacle densities. Obstacle sizes are also chosen randomly using uniform distribution between 0.035 to 0.28 m. Two random seeds are used to generate worlds with $>0\%$ obstacle density. The heuristic weights (eq. 8) were $\omega_{ab} = 7.7$ and $\omega_{abc} = 2.43$ respectively.

All experiments have a randomized initial connector $s_0 = (s_x, s_y, s_z, s_f)$ and goal cube g which defines the goal connectors $\mathbb{S}_g$. We make sure $s_0 \notin \mathbb{S}_g$. We run suites of 100 experiments per world size, obstacle density and obstacle seed. This means data points with 0% obstacles have 100 trials, while others have 200. To run the experiments we used a cluster with Intel Xeon E5-2640 processors running at 2.60GHz. The memory limit by process was always set to 4 GBs and the time limit was 7,200 seconds.

2) Comparing ab and abc Graphs: Figure 8 shows that the double socket *abc* hop model solves significantly more problems than the *ab* approach. This is due to two factors: Firstly, the start and end joint-configurations of the robot when using *abc* are more compact and are less likely to be in collision with an obstacle hovering above the socket. Secondly, not having to fully extend *abc* motions offer higher maneuverability when avoiding obstacles, hence enabling more feasible paths than *ab* which always needs to extend fully between hops.

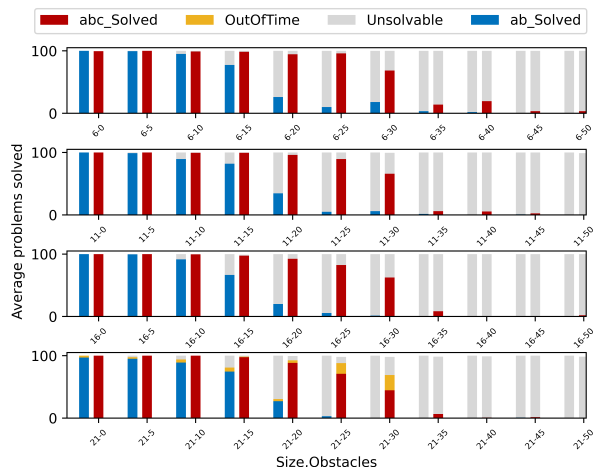


Fig. 8: Average Percentage of Solved, Unsolvable and Out of time outcomes out of trials tested.

Despite *abc*-graph solving more problems, both methods are worth considering as the double socket *abc*-graph relies on a much more expensive cache computation. Also, the two sockets *abc*-graphs are much bigger, see table I.

Lattice	*ab* vertices	*ab* edges	*abc* vertices	*abc* edges
S6O0	216	1056	503	3824
S6O25	82	170	211	940
S6O50	27	10	55	136
S11O0	918	4824	2219	17816
S11O25	386	892	963	4844
S11O50	99	40	222	434
S16O0	2400	13056	5879	48080
S16O25	1023	2532	2649	13948
S16O50	293	234	894	2730
S21O0	4950	27480	12203	100664
S21O25	2058	5096	5389	28312
S21O50	602	444	1570	4316

TABLE I: Vertex and Edge counts of graphs used in testing. Data is for one random seed. The graph sizes shrink as obstacles obstruct sockets or force collision with the robot.

3) Search Effort And Path Costs: Path execution costs over a 100 trials are compared to the search effort spent performing A^* and joint space validation in Fig. 9.

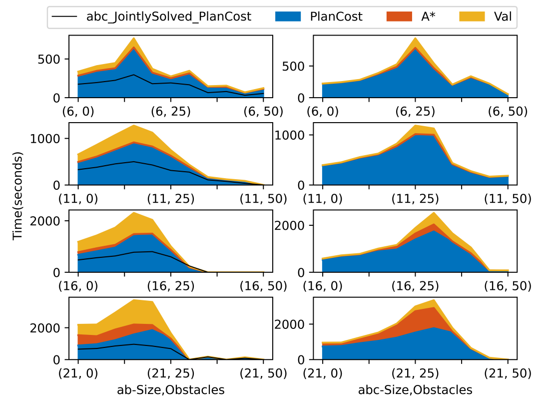


Fig. 9: Stacked average plan execution and search costs (split into $A*$ and validation costs). The black line shows the average plan costs of *abc*-graph only for problems solved by *ab*-graph as well.

As we add obstacles, and hence the graph size shrinks, the robot must perform more complex motions, leading to quickly increasing validation costs. A^* search time increases for larger obstructed lattices as well. In general, to reduce the computational effort, better validation strategies are required. The A^* effort could be reduced by using re-planning algorithms like LPA^*, but that would not affect validation.

The double socket *abc*-graph not only solves all problems that are solved via the single socket *ab* graph, but also finds shorter paths. This is shown via the superimposed black lines denoting *abc* performance on problems solved by *ab*. This is expected as the *ab* motion involves the robot returning to an all-zero joint state. Thus moving longer distances in joint-space and encountering more obstacles. Furthermore, for both graphs, the plan costs increase with obstacle density, but start decreasing as it approaches 50%. This is consistent with the graph size results in Table I as only problems with start and end sockets that are close together are solvable.

4) Time Saved By Replaning: In Fig. 10 we show how re-planning performs, see Sec. V-B.4. In the figure, *savings* mean the difference between the first feasible plan and the best ever found feasible plan. All results are normalized against the first feasible plan cost. For most problems the first found feasible solution is also the best one when taking into account the re-planning search effort. However, actual results are specific to the robot capabilites.

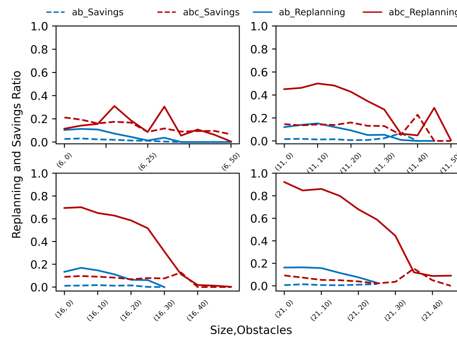


Fig. 10: Average time savings and re-planning effort, in terms of the ratio against the first feasible plan, over all trials.

VII. Conclusions and Future Work

In this paper, we have presented two ways to formulate a graph to perform hybrid discrete-continuous path planning for an inchworm robot on obstructed lattice structures. We showed that for both graphs combining iterative A* search and a pre-computed cache structure scales significantly better than purely online validation strategies.

The double socket *abc* graph formulation clearly outperforms the single socket *ab*, both in terms of problems solved and path costs. Although, the *abc*-graph requires a more expensive offline cache computation, this is reasonable considering target applications are heavy duty tasks like nuclear decommissioning. Furthermore, the offline cache is subject to exploitation of structure-specific symmetries and can be further optimized if limited inverse kinematics solutions are considered combinatorially instead of stochastically.

For both graphs, the search effort is mostly dominated by continuous motion planning costs. Future work could explore optimization-based planners like CHOMP [4] to address this. Additionally, algorithms like LPA* should be used when the search effort is significant compared to the validation effort.

However, the most interesting research direction is to combine robot path planning with structure assembly planning. The ability to accurately estimate traverse costs midway through assembly could prove invaluable for solving the assembly sequencing problem.

VIII. Acknowledgments

This work is partially supported by InnovateUK Connect-R (TS/S017623/1).

References

[1] C. Pan, Y. Han, and J. Lu, “Design and optimization of lattice structures: A review,” *Applied Sciences*, vol. 10, no. 18, 2020.

[2] K. H. Petersen, N. Napp, R. Stuart-Smith, D. Rus, and M. Kovac, “A review of collective robotic construction,” *Science Robotics*, vol. 4, no. 28, 2019.

[3] N. Gershenfeld, M. Carney, B. Jenett, S. Calisch, and S. Wilson, “Macrofabrication with digital materials: Robotic assembly,” *Architectural Design*, vol. 85, no. 5, pp. 122–127, 2015.

[4] S. Leder, R. Weber, D. Wood, O. Bucklin, and A. Menges, “Distributed Robotic Timber Construction,” in *Acadia*, 2019, pp. 510–519.

[5] K. Petersen, R. Nagpal, and J. Werfel, “TERMES: An Autonomous Robotic System for Three-Dimensional Collective Construction,” in *Robotics: Science and Systems VII*, vol. 7, 2011, pp. 257–264.

[6] B. Jenett, A. Abdel-Rahman, K. Cheung, and N. Gershenfeld, “Material–Robot System for Assembly of Discrete Cellular Structures,” *IEEE Robotics and Automation Letters*, vol. 4, no. 4, jul 2019.

[7] B. Jenett and K. Cheung, “BILL-E: Robotic platform for locomotion and manipulation of lightweight space structures,” *25th AIAA/AHS Adaptive Structures Conference, 2017*, pp. 1–12, 2017.

[8] J. Roberts, S. Franco, A. Stokes, and S. Bernardini, “Autonomous Building of Structures in Unstructured Environments via AI Planning,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 31, pp. 491–499, 2021.

[9] H. L. Rodriguez, B. Bridge, and T. P. Sattar, “Climbing ring robot for inspection of offshore wind turbines,” in *Advances in Mobile Robotics*. WORLD SCIENTIFIC, aug 2008, pp. 555–562.

[10] W. Jeon, I. Kim, J. Park, and H. Yang, “Design and control method for a high-mobility in-pipe robot with flexible links,” *Industrial Robot*, vol. 40, no. 3, pp. 261–274, 2013.

[11] G. Paul, P. Quin, A. W. K. To, and D. Liu, “A sliding window approach to exploration for 3D map building using a biologically inspired bridge inspection robot,” *2015 IEEE International Conference on Cyber Technology in Automation, Control and Intelligent Systems, IEEE-CYBER 2015*, no. June 2015, pp. 1097–1102, 2015.

[12] H. Zhang, J. Zhang, G. Zong, W. Wang, and R. Liu, “Sky cleaner 3,” *IEEE Robotics and Automation Magazine*, vol. 13, no. 1, 2006.

[13] K. Kotay and D. Rus, “The Inchworm Robot: A Multi-Functional System,” *Autonomous Robots*, vol. 97, no. 1-4, pp. 131–141, 2000.

[14] B. Jenett, C. Cameron, F. Tourlomousis, A. P. Rubio, M. Ochalek, and N. Gershenfeld, “Discretely assembled mechanical metamaterials,” *Science Advances*, vol. 6, no. 47, pp. 1–12, 2020.

[15] C. H. J. Yang, G. Paul, P. Ward, and D. Liu, “A path planning approach via task-objective pose selection with application to an inchworm-inspired climbing robot,” in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, vol. 2016-Sept, pp. 401–406.

[16] H. Zhu, S. Gu, L. He, Y. Guan, and H. Zhang, “Transition analysis and its application to global path determination for a biped climbing robot,” *Applied Sciences (Switzerland)*, vol. 8, no. 1, jan 2018.

[17] T. L. Lam and Y. Xu, “Climbing strategy for a flexible tree climbing robot - treebot,” *IEEE Transactions on Robotics*, vol. 27, no. 6, pp. 1107–1117, dec 2011.

[18] T. Lam and Y. Xu, “Motion planning for tree climbing with inchworm-like robots,” *Journal of Field Robotics*, vol. 30, no. 1, pp. 87–101, 2013.

[19] D. Pagano and D. Liu, “An approach for real-time motion planning of an inchworm robot in complex steel bridge environments,” *Robotica*, vol. 35, no. 6, pp. 1280–1309, 2017.

[20] D. L. Maria Fox, “The automatic inference of state invariants in tim,” *Journal of Artificial Intelligence Research*, vol. 9, no. 1, pp. 367–421, 1998.

[21] C. Dornhege, P. Eyerich, and T. Keller, “Semantic Attachments for Domain-Independent Planning Systems,” in *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*, 2009, pp. 114–121.

[22] S. Bernardini, M. Fox, D. Long, and C. Piacentini, “Boosting search guidance in problems with semantic attachments,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 27, no. 1, pp. 29–37, Jun. 2017.

[23] S. Koenig, M. Likhachev, and D. Furcy, “Lifelong Planning A*,” *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, may 2004.

[24] A. Stentz, “The Focussed D* Algorithm for Real-Time Replanning,” in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1995, pp. 1652 – 1659.

[25] “Connect-r,” https://www.ross-robotics.co.uk/news/bringing-structure-to-unstructured-environments, accessed: 2022-06-06.

[26] I. A. Sucan and S. Chitta, “Moveit motion planning framework,” 2018.

[27] P. Beeson and B. Ames, “TRAC-IK: An open-source library for improved solving of generic inverse kinematics,” in *IEEE-RAS International Conference on Humanoid Robots*, vol. 2015-Dec. IEEE Computer Society, pp. 928–935.

[28] J. J. Kuffner and S. M. LaValle, “RRT-connect: An efficient approach to single-query path planning,” in *IEEE international conference on robotics and automation (ICRA)*, vol. 2. IEEE, 2000.

[29] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. J. Teller, “Anytime motion planning using the RRT,” in *International Conference on Robotics and Automation, ICRA*. IEEE, 2011, pp. 1478–1483.

[30] I. A. Sucan, M. Moll, and L. E. Kavraki, “The Open Motion Planning Library,” *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, dec 2012.