Applications of Experience Graphs in Humanoid Motion Planning

Sameer Bardapurkar

CMU-RI-TR-17-59

14^{th}August 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Maxim Likhachev (Chair)
Katia Sycara
Sasanka Nagavalli

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

The capability of using cached plans, whether in the form of expert provided demonstrations or previously generated paths from prior planner runs, is a useful addition to any search based planner. Its benefits range from faster planning times in large state spaces to generation of more predictable replans in case of partially known environments which robots discover as they plan. The experience graph (E-Graph) algorithm is a planning algorithm which allows a planner to reuse cached plans/demonstrations by providing a heuristic which biases the search towards states which are present in prior demonstrations. This technical report addresses the particular application of the experience graph algorithm in search based planning for a 35-DOF humanoid. In this work, we outline the details of deploying E-Graphs to this particular problem and show that using E-Graphs significantly outperforms conventional methods like weighted A*.

Contents

List of Figures

Chapter 1

Using Expert Demonstrations for Motion Planning in Humanoids

1.1 Introduction

Humanoid robots have been an active area of research in academia and industry alike. A chief practical advantage that the usage of humanoids provides is that humanoid robots are quite suitable to be used in environments which were designed for human inhabitation. This makes them prime candidates for use in scenarios like search and rescue in industrial environment.

The DARPA robotics challenge of 2015, which saw a great deal of interest by many leading institutions had the primary technical goal which aimed ”to develop human-supervised ground robots capable of executing complex tasks in dangerous, degraded, human-engineered environments”, the fact that a significant portion of the participants in this challenge utilized humanoid robots underlines the relevance of humanoid robots in man-made environments. Additionally, as noted in [1], there exist a lot of areas where humanoid robots can be improved.

WASEDA University, Japan, in collaboration with Mitsubishi Heavy Industries have developed a search and rescue humanoid robot - WAREC-1 (Waseda Rescuer -No. 1). Its a four limbed robot which is capable of tasks such as crawling on uneven terrain such as rubble, climbing ladders and walking on level ground as well as on staircases.

A joint team of CMU and NREC researchers, of which I have been a part of, has been responsible for developing the AI pipeline which will enable WAREC-1 to carry out these tasks. For this purpose, we have utilized a search based planning approach, wherein we plan trajectories for the robot while assuming that the environment is fully known. Since the robot has 28 joints, along with the pose of the torso which is represented as the 3D co-ordinates and the quaternion, the state-space of the robot is 35-dimensional. Computational complexity of planning is thus very high since the state space grows exponentially with the number of dimensions.

1.2 Related Work

In this section I present a brief summary of a few selected works in the area of learning from demonstration in robotics. [2] is a survey paper on the various learning from demonstration techniques used for robots. In the paper, the authors outline two major approaches to learn from an expert provided demonstrations:

- Learn a state-action policy from the provided demonstration: From a demonstration which is categorized by a sequence of states and a sequence of actions, learn a state-action policy through either regression or classification. Once this learning phase is completed, it is then possible for the robot to choose actions based on the states that it encounters.
- Learn an underlying system model from the provided demonstration: From an expert demonstration which is provided by the user, learn the underlying system dynamics (state, action transition probabilities and reward functions). Subsequently, we can solve for a policy from this learned system model and then utilize it during actual operation.

In [3], the authors discuss several trajectory based optimal control techniques. They also highlight the performance of these techniques on real robots such as robotic arms and quadrupedal robots. Some notable techniques which are discussed are Differential Dynamic Programming and Path Integral Reinforcement Learning. In [4] , the authors use learning from demonstration to carry out a task of swinging up a pendulum. They first learn a reward function from the demonstrations and then proceed to generate a task model by repeatedly trying to perform the task. As the dimensionality of the problem increases, so does the complexity of solving it - known as the ”curse of dimensionality”. In [5], the authors try to address this problem by representing value functions and policies non-parametrically along a set of previously generating trajectories. By doing so across trajectories, they hope to create a global value function or policy. Learning from demonstration framework is leveraged in the field of control in [6], where the authors start with a simple linear dynamical model of a system and transform it into a non-linear model by introducing a learn-able term. They then use demonstrations to learn this augmented model. The advantages of such learning based methods is that they do not require an underlying system model since they are end-to-end methods. However, they do require a high number of training examples, which may need to long training times, especially for higher dimensional planning problems.

Aside from learning based methods, there has also been a lot of work in creating databases or libraries of trajectories and then using this information at plan time. The authors in [7] use such an approach to achieve a standing balance control for a two link manipulator. Using libraries of optimal trajectories and a neighborhood optimal control method, the authors were able to make this two link manipulator withstand pushes while being in a vertical state. The approach of learning policies based on demonstrations is extended to a library of trajectories in [8] where the authors aim to learn a policy using not just value functions but also by encouraging the use of the trajectories in the database. They further extend this framework into a framework which allows such policies to be transferred across tasks and environments, as highlighted in [9]. Also worth mentioning is [10] which outlines several techniques to prune trajectories from a database of trajectories.

On the planning side, there has been a lot of work on using previously generated plans in sampling based planners. [11] is a work which introduces an extension of Rapidly-Exploring Random Trees (RRTs) called execution extended RRT (ERRT). In this algorithm, the authors develop a technique for fast re-planning by caching successful plans and biasing future plans towards the way points of these caches. An extension of this approach is [12] where the authors address the problem of planning in dynamic environments. In such a scenario, they encourage the planner to reuse as much of the plan as possible. Another such extension can be found in [13] which is also intended for use in dynamic environment.An example which uses sampling based planning techniques with trajectory libraries can be found in [14]. This method works by extracting certain key points from previously generated plans and then biasing a sampling based planner towards sampling points around these key-points. However, this method is limited to reusing only one previous plan at a time. The authors in [15] cache previously known plans and also certain feature vectors which depict the environment under which the plans were made. During plan time, these learned features are then used to map to a suitable plan in this database of plans. In search based planning, one approach of using previously known plans is Experience Graphs (E-Graphs), highlighted in [16].Experience graphs allow us to reuse past plans by generating a heuristic which biases the search towards segments of the previously made plans. The heuristic, called E-Graph heuristic is computed as per Eq. 1.1. To calculate the heuristic value for a state s_0, we calculate the cost of the minimum cost sequence of segment from all possible sequence of segments π, where $\pi =< s_0 ... s_{N-1} >$ such that S_0 is the start state and S_{N-1} is the goal state. The segment cost in a sequence of segments is computed as the minimum of the original heuristic (h^G) inflated by a factor ϵ^E and the cost of traversing that segment using edges of the experience.

$$h^E(s_0) = \min_{\pi} \sum_{i=0}^{N-1} \min\{\epsilon^E h^G(s_i, s_{i+1}), c^E(s_i, s_{i+1})\} \tag{1.1}$$

1.3 Problem Formulation

As noted in [17], discrete representation of states is a well-established method of reducing the computational complexity of a motion planning problem. Borrowing from this idea, we represent the state space of the humanoid robot $S_{humanoid}$, as a set of discrete full body configurations of the humanoid. Since the humanoid state is represented in 35 dimensional form, $S_{humanoid} \subseteq \Re^{35}$. We thus formulate the motion planning problem as a search on a discrete graph $G(V^G, E^G)$ where $V^G \subseteq \Re^{35}$ is the set of discrete configurations of the humanoid and E^G are the edges connecting pairs of vertices in V^G. Each $e \in E^G$ corresponds to the result of an action, known as a motion primitive, which when a applied to a vertex $v \in V^G$ results in the humanoid transitioning from one configuration to another. Motion primitives are discussed in greater detail in 1.4.2. Additionally, we also need a notion of cost of executing a particular motion primitive. For the purposes of this problem, a cost is simply calculated as the total sum of the joint angles moved by the corresponding motion primitive.

We specify our planning problem as the one where we have to find a valid sequence of actions, which will transition the humanoid from a start configuration $s_{start} \in V^G$ to a goal configuration $s_{goal} \in V^G$. In the context of this problem, a valid action or a configuration is one which is within the joint limits of the humanoid, is free from collisions and is statically stable. More details about checking for validity can be found in a subsequent section. Thus, to summarize, we need to find a set of valid states π such that,

$$\begin{aligned} \pi &=< s_0, s_1, ..., s_N > \\ s_0 &= s_{start} \\ s_N &= s_{goal} \\ s \in V^G \quad \forall \quad s \in \pi \\ (s_i, s_{i+1}) \in E^G \quad \forall \quad (s_i, s_{i+1}) \in \pi \end{aligned} \tag{1.2}$$

The planner has to operate in the full body joint space since it is not possible to abstract away any degrees of freedom by making assumptions about the underlying structure of the problem. This severely increases the computational complexity of the planning problem since not only is the state-space 35-dimensional, but the start and goal configurations are precise 35 dimensional configurations which need to be exactly met - we do not have a goal region where we can plan to. Additionally, validity checks in full dimensional space are costly, making implicit graph generation a time consuming affair.

In order to solve this challenge of computational complexity, we can leverage two observations:

- The transitions are more or less regular, like for example, transitioning from crawling to bipedal representation.
- The environment will mostly be the same, for example, the shape and structure of the ladder.

In this work, we use E-Graphs [16] along with expert generated demonstrations to build this planner.

1.4 Methodology

1.4.1 Generating expert demonstrations

The E-Graph planner makes use of previously generated full or partial plans in the current planning iteration. While it is possible to cache plans made by this planner and reuse them in upcoming planning iterations, it would be beneficial to have a framework to generate these demonstrations in the first place. For this purpose, I have developed a tool which will allow a user to manually generate valid demonstrations for accomplishing plans from start to goal.

Through this tool, the user can load various environment meshes (in `.stl` format), the user can then load the robot in any desired full body configuration and manipulate its state. The state of this

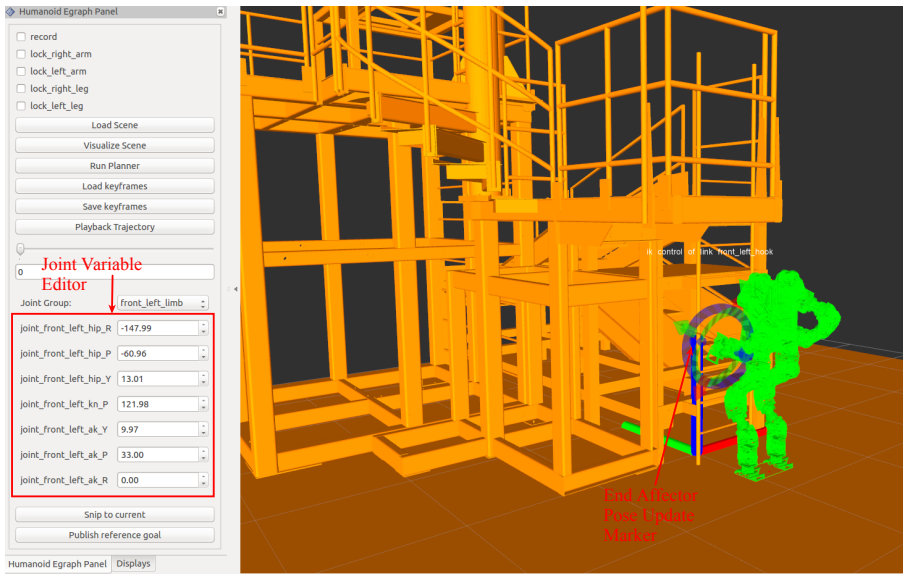


Figure 1.1: Tool to generate valid demonstrations for the E-Graph planner

robot can be changed by the user by either changing a joint angle at a time (which corresponds to making the demonstration interface carry out forward kinematics (FK) computations to compute the overall poses of all the robot frames), or by altering the pose of one of the four end effectors of the robot (which corresponds to the demonstration interface carrying out inverse kinematics (IK) computations to update the affected joint angles of the corresponding joint groups). This tool is a GUI which can be added as a panel in `rviz`, the de-facto visualizer used in ROS.

This tool uses the same validity checker as the planner uses. So any demonstration made by this tool in a given environment is guaranteed to be valid when the planner is running in the same environment. The only time when segments of demonstrations become invalid in the planner is when the planner is running in a different environment as the one which was present when the tool was used, and thus some segments of the demonstration may be invalidated by the presence of obstacles. The demonstration tool is shown in Fig. 1.1.

The demonstrations themselves comprise of full joint space key-frames, where each key-frame is a $1X35$ length vector. They are stored in a comma separated value (`.csv`) file format with a header which records the joint names.

1.4.2 Planner Details

Heuristic Computation

As discussed in Eq. 1.1 an E-Graph heuristic for a state so, $h^E(s_0)$ on a graph G, with an experience graph of E is calculated as $h^E(s_0) = \min_\pi \sum_{i=0}^{N-1} \min\{\epsilon^E h^G(s_i, s_{i+1}), c^E(s_i, s_{i+1})\}$.

In order to make the heuristic computation tractable in the high dimensional state-space of the humanoid, the heuristic is computed in (x, y, z) space. This is achieved by projecting the experience graph by projecting each full dimensional state in the experience graph into the (x, y, z) space. This projection is achieved by the $projectToXYZ$ function. Depending on the requirements, the $projectToXYZ$ function can be implemented in different ways. For the purposes of the humanoid E-Graph planner, the $projectToXYZ$ function projects a full body humanoid state to the (x, y, z) co-ordinates of a particular end-effector of a robot. In order to perform heuristic pre-computations,an input demonstration is projected into this space using the $projectExperienceGraph$ procedure in Algorithm 1. This procedure projects valid vertices and edges of the demonstrations into the lower dimensional space. Once the E-Graph is projected into the lower dimensional space, I precompute the heuristic by propagating a 26-connected 3D-Dijkstra grid from the goal. In this grid, the costs of edges which are not part of the demonstration are penalized by factor of ϵ^E whereas the costs of edges which are part of the demonstration are not penalized. To maintain computational efficiency, this grid is constructed on-demand. The grid frontier is only propagated until the node for which the heuristic is queried is expanded. Once this is done, the grid is not propagated any further and the frontier is preserved. When another query is made, this frontier is re-expanded. This is the $computeHeuristic$ function of Algorithm 1. In the algorithm, $OPEN$ is a priority queue which assigns priority to states based on their g values.

For the planner, I maintain four such E-Graph Dijkstra grids, with each grid projecting a full body configuration to the (x, y, z) co-ordinates of each of the end-effectors of the robot. The final value of the heuristic for a full body configuration of the robot is simply the sum of the four individual heuristics. Fig. 1.2 shows the contour mapping for the E-Graph Dijkstra heuristic for the right hand of the robot for a goal on the ladder. The key stages of the demonstration are shown in Fig. 1.2a through Fig. 1.2d. As can be noticed, the heuristic values (red is a bad value and blue is a good value), encourage the right hand of the robot to follow the path to the goal (on the ladder) as close as possible to the originally demonstrated path.

In this case, the $projectToXYZ$ function projects a full body humanoid state to a corresponding end-effector state by carrying out the forward kinematics for the kinematic chain to which that particular end-effector belongs. For these purposes, we can model the humanoid as four open chained manipulators joined to a single root which in this case is the torso. Thus, a particular end-effector can be projected to XYZ dimensions by running an open chain forward kinematics solver, which solves equations like the one shown in Eq. 1.3. This kind of forward kinematics solver is henceforth represented by $\lambda^{FK}(i, s)$ in Eq. 1.4. Which takes as input the id of the end-effector, i and the full body joint state $s \in S_{humanoid}$. As an output, it returns the pose of that end-effector in the global frame. This pose is represented by the 3D co-ordinates and the quaternion depicting the orientation of the end-effector.

$$[T] = [Z_1][X_1][Z_2][X_2]...[X_6][Z_7] \tag{1.3}$$

$$\lambda^{FK}(i, s) = [x_i, y_i, z_i, q_i^x, q_i^y, q_i^z, q_i^w] \tag{1.4}$$

In Eq. 1.3, $[T]$ is the pose of the end effector, $[Z_i]$ denotes the transformation due to the i^{th} joint and

X_i is the rigid transformation corresponding to the dimension of the i^{th} link. For implementation purposes, I use the inbuilt forward kinematic solvers provided by `ROS`.

A typical use case of this heuristic is initiated by a call to the $Initalize$ procedure, which begins by setting the value of ϵ^E (Line 10) and initializes the Dijkstra frontier and states. It then calls the e$projectExperienceGraph$ procedure, which then projects the vertices V and edges E of the expert demonstration to $x-y-z$ space by using the forward kinematics of the robot. Before exiting, the $Initialize$ procedure initiates the frontier of the Dijkstra grid by seeding it with the goal state. During plan time, when we want to query for the value of heuristic for a state s_{query}, we call the $computeHeuristic$ function. This function propagates the Dijkstra frontier until the s_{query} state is expanded. This function begins by projecting s_{query} into $x-y-z$ space, Line 23. It then proceeds to expand the Dijkstra frontier until we have found the cost of a least cost path of reaching s_{query} from s_{goal}. This is shown in Line 24 to Line 30. In order to update the minimum cost of reaching s from s_{goal}, which is the $g-value$ of s, we use the $computeKey$ function. This function updates the $g-value$ of a state s' if its cheaper to reach s' from the goal by taking a path through a state s''. The edge cost is computed according to the specifications of E-Graphs, where edges on the demonstrations are not penalized and edges off the E-Graphs are penalized by multiplying them with a factor ϵ^E, as shown in Line 18.

Motion Primitives

As noted in 1.3, the problem of motion planning for humanoid has been reduced to a search problem on a graph $G(V^G, E^G)$. Further on, in 1.3 we also discussed the concept of a motion primitive which is responsible for creation of an edge $e \in E^G$. In this section, we will formally define the notion of a motion primitive.

A motion primitive is a pre-defined action which when applied to a full body state $v_{in} \in V^G$ results in a full body state $v_{out} \in V^G$. Thus, in essence, it creates an an edge $e \in E^G$ which connects v_{in} and v_{out} subject to the condition that the transition $v_{in} \rightarrow v_{out}$ is valid. Validity checking is discussed in detail in 1.4.2. We use the following motion primitives for the humanoid:

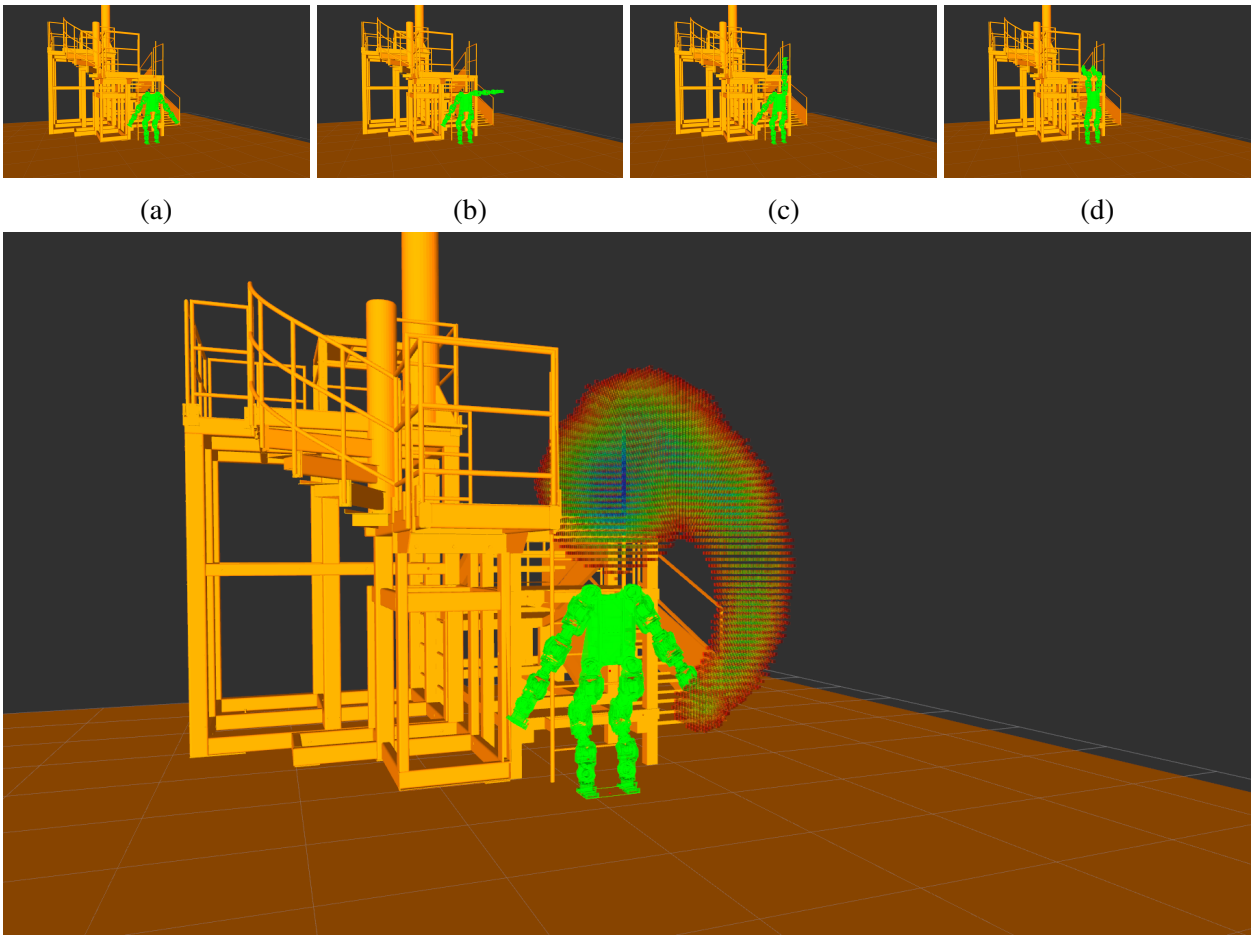
- Joint Space Primitives : These motion primitives correspond to changing one or many joint variables at a time. For example, a joint space motion primitive may involve changing the i^{th} joint variable of the robot by a value δ_i. In such a case, if the input state a is given by:

$$a = [a_0, a_1, ..., a_i, ..., a_{34}] \tag{1.5}$$

  The application of such a motion primitive will result in a state b given by,

$$b = [a_0, a_1, ..., a_i + \delta_i, ..., a_{34}] \tag{1.6}$$

- Limb Cartesian IK Primitives : These motion primitives correspond to a metric translation/rotation of a given end-effector at a time. Once the new pose of the end-effector is set, an Inverse Kinematics (IK) solver then recomputes the joint angles of the relevant joint



(a) (b) (c) (d)

(e)

Figure 1.2: Fig. 1.2e illustrates the right hand heuristic for the demonstration depicted in Fig. 1.2a through Fig. 1.2d. Values are color coded from blue to red.

Algorithm 1 DijkstraEGraphHeuristic3D

procedure PROJECTEXPERIENCEGRAPH([V,E])
  $V_{egraph} \leftarrow \emptyset$
  $E_{egraph} \leftarrow \emptyset$
  for $v \in V$ **do**
    if valid(v) **then** V_{egraph}.insert(projectToXYZ(v))
  for $e \in E$ **do**
    if valid(e) **then** E_{egraph}.insert(projectToXYZ(e))
end procedure
procedure INITIALIZE(s_{goal}, $penalty_{egraph}$)
  $\epsilon^E \leftarrow penalty_{egraph}$
  $OPEN \leftarrow \emptyset$
  $CLOSED \leftarrow \emptyset$
  projectExperienceGraph()
  $s_{goalxyz}$ = projectToXYZ(s_{goal})
  $OPEN$.insert($[s_{goalxyz}, 0]$)
end procedure
function COMPUTEKEY(s', s'')
  $c(s'', s') = \min[\epsilon^E c^G(s'', s'), c^E(s'', s')]$
  return $\min[g(s'), (g(s'') + c(s'', s'))]$
end function
function COMPUTEHEURISTIC(s_{query})
  $heuristic \leftarrow 0$
  s_{xyz} = projectToXYZ(s_{query})
  while $s_{xyz} \notin CLOSED$ **do**
    $s = OPEN$.TopKey()
    $CLOSED = CLOSED \cup s$
    for $s' \in$ neighbors(s) **do**
      if $s' \notin CLOSED$ **then**
        $g(s')$ = computeKey(s')
        $OPEN$.insert($[s', g(s')]$)
  $heuristic = g(s_{query})$
  return $heuristic$
end function

groups. For example, if a motion primitive which corresponds to moving an end-effector i along the x-axis by an amount δ_x is applied to an input state a as defined above, it will result in an output state b such that,

$$\begin{aligned} \lambda^{FK}(i,a) &= [x_a, y_a, z_a, q_a^x, q_a^y, q_a^z, q_a^w] \\ \lambda^{FK}(i,b) &= [x_a + \delta_x, y_a, z_a, q_a^x, q_a^y, q_a^z, q_a^w] \end{aligned} \tag{1.7}$$

If no feasible IK solution is found, then the corresponding motion primitive does not result in a successor state. In order to calculate IK, I use the `trac-IK` package of `ROS` which solves for IK through a combination of joint-limited pseudoinverse Jacobians and a Sequential Quadratic Programming IK formulation that uses quasi-Newton methods.

- Torso IK Primitives : The motion primitives correspond to changing the pose of the torso of the robot. Once the new pose of the torso is set, an Inverse Kinematics (IK) solver then runs to recompute the joint angles such that the poses of all the end-effectors remain unchanged. This corresponds to the robot moving its torso while not moving its hands or legs. For example, if a torso primitive which corresponds to moving the torso along the x-axis by a value of δ_x is applied to a full body state a as defined above, and if a_0 encodes the x co-ordinate of the torso, then the application of this primitive will result in a state b such that,

$$b = [a_0 + \delta_x, b_1, ..., b_{34}] \tag{1.8}$$

Such that for any end affector i,

$$\lambda^{FK}(i,a) = \lambda^{FK}(i,b) \tag{1.9}$$

Validity Checks

Once we apply motion primitives to the a particular state, we need to run validity checks to ensure that the corresponding edge that has been added to the search graph is valid. We carry out the validity checks as follows:

1. Linearly interpolate along the edge to create intermediate states between the states for which the validity is to be checked : In order to validate a transition between two states a and $b \in S_{humanoid}$, we need to not only check the validity of a and b but also the validity of the states we might encounter as we transition from a to b. There are many methods to estimate which intermediate states we may encounter, and for the purposes of this paper we simply linearly interpolate between state a and state b with a fixed resolution r. Thus, we carry out state validity check on the set of all intermediate states S_{interm} which lie between a and b. A state $s \in S_{interm}$ is defined as,

$$\begin{aligned} &s = [\min(a_0 + t\delta_0, b_0), \min(a_1 + t\delta_1, b_1)...., \min(a_{34} + t\delta_{34}, b_{34})] \\ &t \in \mathbb{N} \\ &\delta_i = \begin{cases} r, & \text{if } b_i \geq a_i \\ -r, & \text{otherwise} \end{cases} \end{aligned} \tag{1.10}$$

2. Run state validity checker on the interpolated states: This corresponding to checking an intermediate state $s \in S_{interm}$ for validity.

For every state $s \in S_{interm}$, the state validity checker checks the following:

1. Static stability check : We first compute the 3D co-ordinates of the center of mass of the robot in state s. This can be trivially computed since we know the full configuration of the robot and the masses of its links and joints. Let the co-ordinates of the center of mass be $[x_{COM}, y_{COM}, z_{COM},]$. Now, we run $\lambda^{FK}(i, s)$ for all end effectors of the robot. This gives us the world poses of the end effectors. Based on these poses, known geometry of the end effector and the knowledge of the location of support surfaces of the environment, it is now possible for us to calculate the convex hull of all the end effector points which are in contact with the support surfaces or the ground. We project this convex hull on the $x - y$ plane and call it the "support polygon" (sp). If the $x - y$ projection of the center of mass $[x_{COM}, y_{COM}]$ lies inside sp, then we say that the state s is statically stable. If the state is not statically stable, the validity checker returns false.

2. Collision checking : Collision checks are performed by abstracting the robot by a set of collision spheres $S_{collision}$, which allow for faster collision checking. The collision spheres for the humanoid are shown in figure Fig. 1.3. Based on the known environmental obstacles, we construct a distance field $F(x, y, z)$ such that,

$$F(x, y, z) = \text{distance of nearest obstacle from the point}[x, y, z] \tag{1.11}$$

   In practice, this can be achieved by propagating a breadth first search from all of the occupied 3D voxels. To collision check the robot, we then check each sphere for collision against another sphere and collision against an obstacle. Thus, for a sphere $s^i \in S_{collision}$ we have its center $c^i = [c^i_x, c^i_y, c^i_z]$ and its radius r_i. We report an environmental collision if:

$$F(c^i_x, c^i_y, c^i_z) \leq r_i \tag{1.12}$$

   Additionally, we also check for self collision, which occurs for spheres $(s^i, s^j) \in S_{collision}, (i \neq j)$ when,

$$\|c^i - c^j\| \geq r^i + r^j \tag{1.13}$$

   Since the spheres are an abstract representation of the robot itself, it may be possible that spheres which are adjacent to each other overlap. Thus, we maintain a list of pairs of spheres $(s^i, s^j) \in S_{collision}, (i \neq j)$ which we expect will overlap. We only check self collision for spheres which are not part of this list.

3. Sliding checks : Is the robot dragging any end-effector across a support surface when transitioning between states. This is carried out by maintaining a flag for each state $s \in S_{interm}$ which checks if an end-effector i contacts a support surface or not. This can be checked by running $\lambda^{FK}(i, s)$ and checking against the location of support surfaces. if for consecutive states $s_a \rightarrow s_{a+1}, s_a, s_{a+1} \in S_{interm}$ we find that an end effector i is in contact for s_a and s_{a+1} and if $\lambda^{FK}(i, s_a) = \lambda^{FK}(i, s_{a+1})$, then we determine that a sliding has taken place and we invalidate the transition.

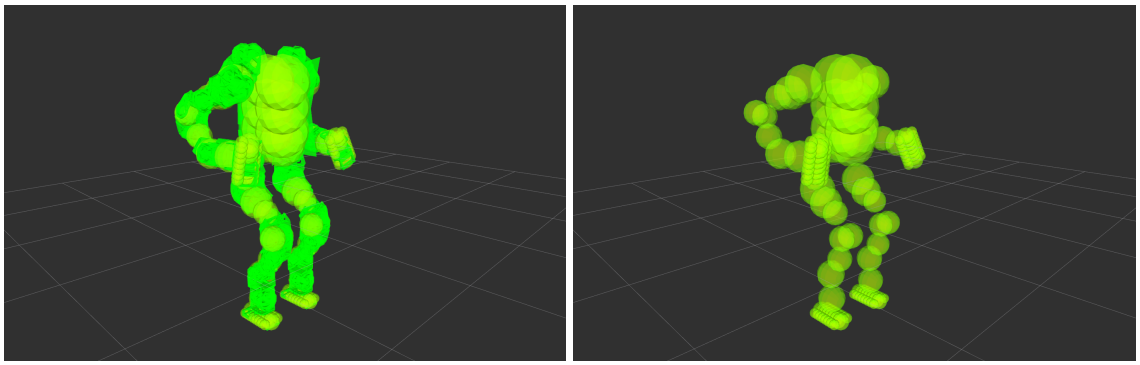


Figure 1.3: The collision spheres of the humanoid robot

4. Ground intersection check : For every state $s \in S_{interm}$, and every end effector i, we run $\lambda^{FK}(i, s)$. If we find that the z co-ordinate of that end effector is negative, we return a ground collision and invalidate the transition.

Generating Shortcuts

In [16], the authors introduce two additional action types apart from motion primitives:

- Snap Successors : These successors are generated when we construct an edge between a node on the search graph and a node on the expert demonstration.
- Shortcut Successors : These successors are generated when we expand a node which is present on the expert demonstration, and we leverage the demonstration itself to jump to a state near the goal without having to re-generate paths in the search graph.

These two kinds of actions are critical for the humanoid planner since they allow for quick snaps to the goal. However, there is a catch, due to the nature of the humanoid, it may not always be possible to fully snap on to a demonstration state. Consider an example case where the demonstration provides a set of nodes which guide the arms and hands towards grabbing on to a ladder rung. Additionally, consider that during planning we use this demonstration. However, the location of the feet during plan time differs from the location of feet in the demonstrations. Thus, it is hard for the planner to fully snap on to the demonstration without first applying actions which step the feet on to the same feet locations as those in the demonstration. This is a complicated method to snap on to a demonstration since it requires the planner to precisely execute footsteps. However, we can note that if the demonstration footstep location and the planner footstep location is close enough, it is possible to move the torso to the same location as that in the demonstration while keeping the feet grounded, and then snap on the arms to the demonstration and follow the demonstration to grab on to the ladder rung. Thus, in effect, we are partially snapping to the demonstration state and to the shortcut. If the demonstration is represented as a set of nodes on the graph, then this essentially translates into generating an adaptive motion which consists of nodes present in a tunnel

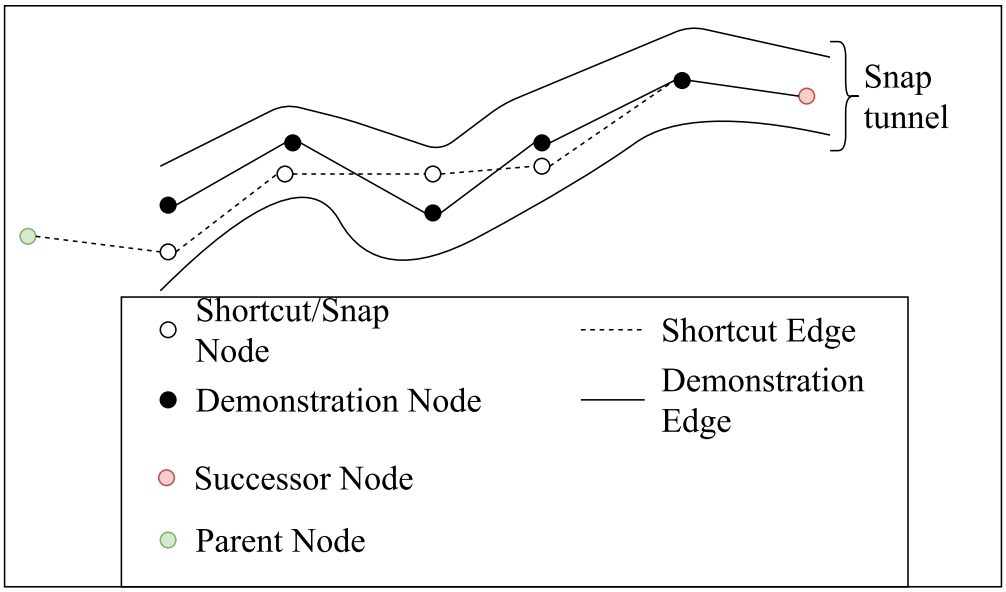


Figure 1.4: Generating snap motions and shortcuts for the humanoid

around the original demonstration. This partial snapping is shown in 1.4. In order to achieve this kind of behavior in my algorithm, I use the $snap$ function as illustrated in Algorithm 2. The $snap$ function takes as input two states, the initial state and the target state to snap to. It then tries to snap the initial state as best as it can to the target state, while ensuring that end-effectors of the robot which were in contact with ground or support surfaces in the initial state are either not in contact in the snap state, or if they are in contact in the initial state, then their pose in the snap state is the same as their pose in the initial state. This ensures that the robot does not drag or slide its end-effectors.

Shortcut successors are then generated by repeatedly calling $snap$ along the original demonstration, and generating an adaptive shortcut in a tunnel around the original demonstration. It may be possible that along this shortcut it is possible to fully snap on to the demonstration (as seen in 1.4).

Search Algorithm

Since the humanoid may need various heuristics to make a feasible plan from a generic start to a generic goal location, I have used the Multi Heuristic A* (MHA*) algorithm as detailed in [18]. Additionally, I have implemented a version which does lazy edge evaluations. Thus, if a state s generates n successors, then the edges from s to each of those n successors are assumed to be valid until one of those successors is chosen for expansion. It is at this time when we check if the edge is valid. If it is not, then we simply proceed to the next state in $OPEN$. Using this planner significantly increases planning speed since validity checks are costly.

Algorithm 2 getSnapSuccessor

```
1: function SNAP(s_init, s_target)
2:     s_snap ← s_target
3:     for end_affector ∈ ROBOT_END_AFFECTORS do
4:         init_hook_pose = GetEndEffectorPose(s_init, end_affector)
5:         snap_hook_pose = GetEndEffectorPose(s_snap, end_affector)
6:         if init_hook_pose ≠ snap_hook_pose then
7:             if InContact(init_hook_pose) ∧ InContact(snap_hook_pose) then
8:                 SetLinkPose(s_snap, end_affector, init_hook_pose)
9:     if IsValidState(s_snap) then
10:        return s_snap
11:    else
12:        return s_init
13: end function
14: function GETSHORTCUT(s_graph , demonstration)
15:    shortcut ← ∅
16:    s_init ← s_graph
17:    for s_demo ∈ demonstration do
18:        if s_init ∉ shortcut then
19:            shortcut = shortcut ∪ s_init
20:        s_snap =snap(s_init, s_demo)
21:        s_init = s_snap
22:    return shortcut
23: end function
```

1.5 Experimental Results

This section details the experiments carried out on the humanoid.

1.5.1 Test Environment

The test environment is shown in Fig. 1.5a. A typical start state of the robot is shown in Fig. 1.5b and the goal state of the robot is a configuration in which the robot is mounted on the ladder(as shown in Fig. 1.5c). For test purposes, I test the robot in start states which are close to this indicated start state, but is offset from the initial state of the demonstration by a distance in the range of 0-5 cm along the $x - axis$. This is done to test the robustness of the planner and its capabilities to adaptively come up with a path.

1.5.2 Generated Demonstration

For the E-Graph planner I have constructed a demonstration using the demonstration tool. The demonstration is shown in Fig. 1.6. The start state of the demonstration is different from the start state of the planner. Additionally, some states of the demonstration are not valid in the current environment (shown in red). This is intentional and is meant to show the path repair capabilities of the planner. The demonstration makes the robot raise both arms, grip the ladder with its left hand, then with its right and and subsequently placing both its feet on the ladder.

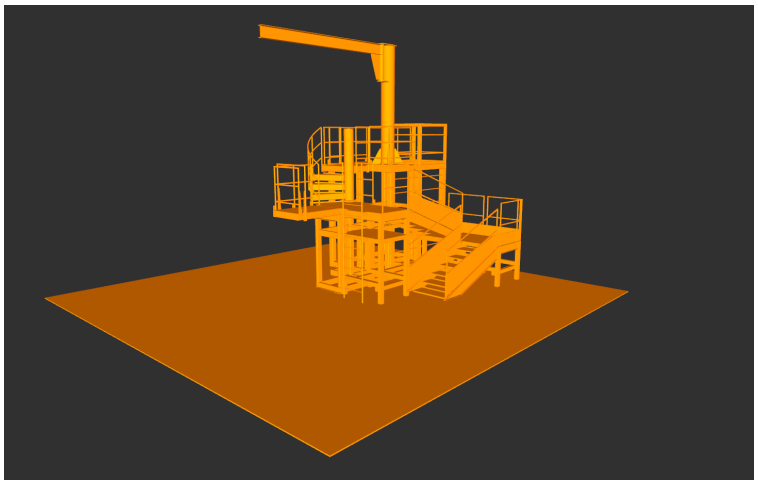
1.5.3 Results and Discussion

Test results are in Table 1.1. All results are averaged over 50 runs. As can be seen, the computational effort and time consumption increases when the start state is offset from the demonstration. From the results, we can make the following inferences:

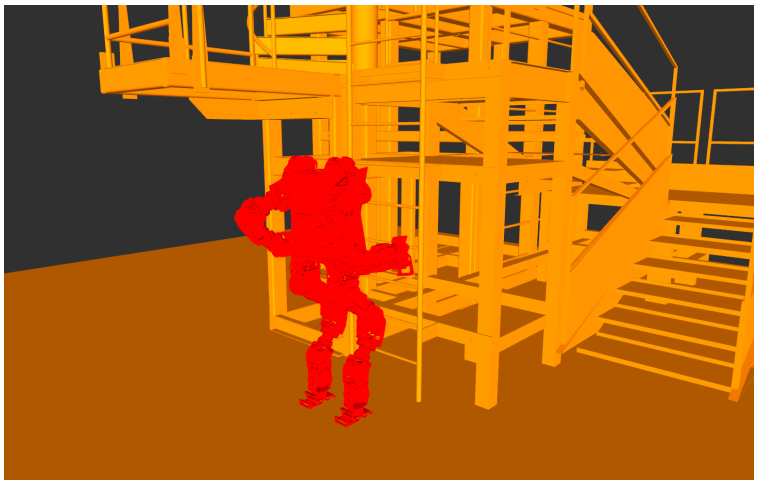
Δx	Time (s)	Expands	% Expands on path
0.0	21.74	50	67
0.03	55.7	79	47

Table 1.1: Test results of the Humanoid E-Graph Planner

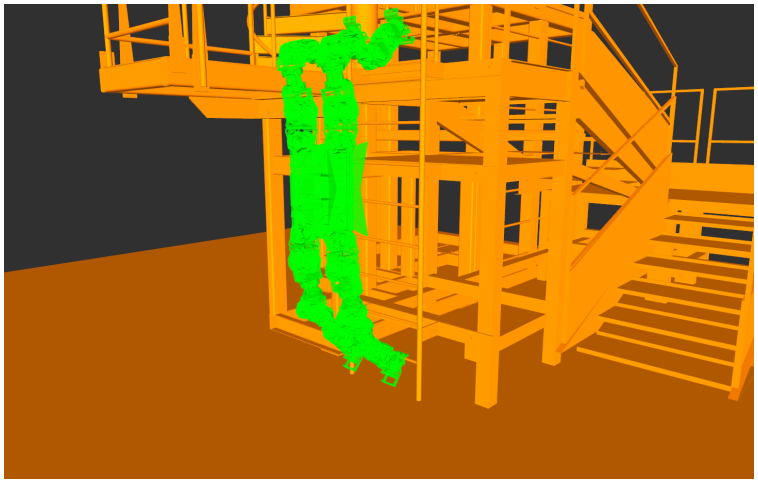
- The tests where the robot stands at the same location as the demonstration start point take lesser time to run than the tests in which the start state is offset from the demonstration
- On an average, 67% of all the expands made by the planner in the zero offset case are states which are also on the path that the planner found. This is an indication that the effort expended by the planner is focused on repairing the broken demonstration.



(a) Enviroment used for the tests



(b) Start State of the robot



(c) Goal State of the robot

Figure 1.5: Test Setup

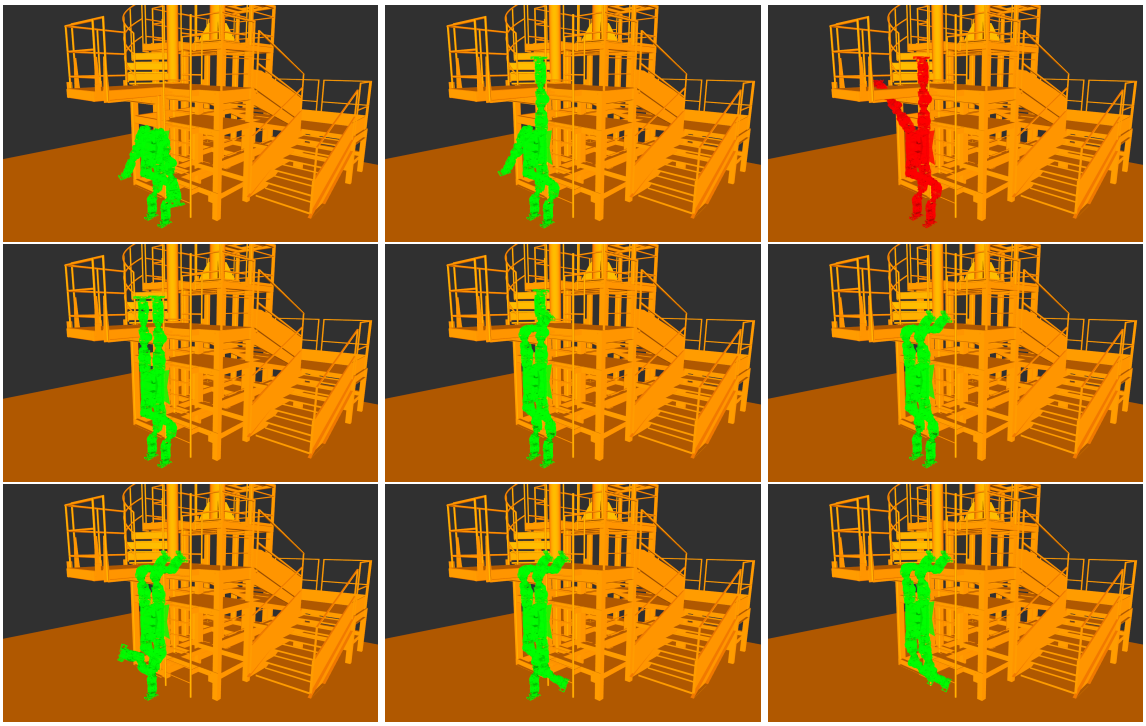


Figure 1.6: Demonstration

- On an average, 47% of all the expands made by the planner in the case where start states are offset from the demonstration start state are states which are also on the path that the planner found. This is an indication that the effort expended by the planner is focused on repairing the broken demonstration and also to carry out some exploration.

When we tried to run common search algorithms like weighted A*, we failed to get a feasible solution within a timeout of 5 minutes.

1.6 Conclusions and Future Work

In this work we outline a framework of using expert demonstration within the purview of motion planning for a humanoid robot. We describe the methodology of E-Graphs which we have leveraged to solve this problem and we then describe the nuts and bolts of how certain key implementation details. We also outline the preliminary results obtained by testing this framework in the context of a ladder mounting task of a humanoid. We have shown that this algorithm is capable of repairing an expert demonstration and generating plans to successfully achieve this task and significantly outperforms conventional search based planners like A*.

However, the testing of this algorithm remains limited. In the future, we would like to carry out a far more rigorous testing of this algorithm. Additionally, we would also like to extend this frame-

work by adding a capability which can integrate multiple demonstrations and feed back successful plans to add to a database of demonstrations. Lastly, a smarter way of storing demonstrations, by extracting features, for example, is also a worthwhile direction to investigate.

Bibliography

[1] C. G. Atkeson, B. Babu, N. Banerjee, D. Berenson, C. Bove, X. Cui, M. DeDonato, R. Du, S. Feng, P. Franklin, *et al.*, "What happened at the darpa robotics challenge, and why," *submitted to the DRC Finals Special Issue of the Journal of Field Robotics*, 2016. 1.1

[2] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009. 1.2

[3] S. Schaal and C. G. Atkeson, "Learning control in robotics," *IEEE Robotics & Automation Magazine*, vol. 17, no. 2, pp. 20–29, 2010. 1.2

[4] C. G. Atkeson and S. Schaal, "Robot learning from demonstration," in *ICML*, vol. 97, pp. 12–20, 1997. 1.2

[5] C. G. Atkeson and J. Morimoto, "Nonparametric representation of policies and value functions: A trajectory-based approach," in *Advances in neural information processing systems*, pp. 1643–1650, 2003. 1.2

[6] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical movement primitives: learning attractor models for motor behaviors," *Neural computation*, vol. 25, no. 2, pp. 328–373, 2013. 1.2

[7] C. Liu and C. G. Atkeson, "Standing balance control using a trajectory library," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pp. 3031–3036, IEEE, 2009. 1.2

[8] M. Stolle and C. G. Atkeson, "Policies based on trajectory libraries," in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pp. 3344–3349, IEEE, 2006. 1.2

[9] M. Stolle, H. Tappeiner, J. Chestnutt, and C. G. Atkeson, "Transfer of policies based on trajectory libraries," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pp. 2981–2986, IEEE, 2007. 1.2

[10] M. S. Branicky, R. A. Knepper, and J. J. Kuffner, "Path and trajectory diversity: Theory and algorithms," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pp. 1359–1364, IEEE, 2008. 1.2

[11] J. Bruce and M. Veloso, "Real-time randomized path planning for robot navigation," in *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, vol. 3, pp. 2383–2388, IEEE, 2002. 1.2

[12] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with rrts," in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pp. 1243–1248, IEEE, 2006. 1.2

[13] M. Zucker, J. Kuffner, and M. Branicky, "Multipartite rrts for rapid replanning in dynamic environments," in *Robotics and Automation, 2007 IEEE International Conference on*, pp. 1603–1609, IEEE, 2007. 1.2

[14] X. Jiang and M. Kallmann, "Learning humanoid reaching tasks in dynamic environments," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pp. 1148–1153, IEEE, 2007. 1.2

[15] N. Jetchev and M. Toussaint, "Trajectory prediction: learning to map situations to robot trajectories," in *Proceedings of the 26th annual international conference on machine learning*, pp. 449–456, ACM, 2009. 1.2

[16] M. Phillips, B. J. Cohen, S. Chitta, and M. Likhachev, "E-graphs: Bootstrapping planning with experience graphs.," in *Robotics: Science and Systems*, vol. 5, 2012. 1.2, 1.3, 1.4.2

[17] M. Pivtoraiko and A. Kelly, "Efficient constrained path planning via search in state lattices," in *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, pp. 1–7, 2005. 1.3

[18] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic a*," *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 224–243, 2016. 1.4.2