TIME ORIENTED ANYTIME MOTION PLANNING WITH TEMPORAL DIFFERENCE LEARNING

A Thesis

by

MATTHEW DAVID GROGAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Stavros Kalafatis
Co-Chair of Committee,	Dylan Shell
Committee Member,	I-Hong Hou
Head of Department,	Miroslav Begoric

May 2019

Major Subject: Computer Engineering

ABSTRACT

Anytime algorithms are a class of algorithm which are interruptible and whose solution quality improves with time, tending towards an optimal solution. In other words, there is a non-decreasing relationship between time invested in computation and solution quality. Algorithms of this nature are clearly relevant to the problem of robotic motion planning. When the state space is large or high-dimensional as is the case for many real applications, an optimal trajectory may take prohibitively long to compute. Using an anytime approach allows for a sub-optimal yet feasible solution to be returned in a reasonable amount of time, after which further time can be spent on improvement. When the nature of this improvement is defined by something like path length or mechanical work, the trade off between time and solution quality must be engineered for a specific context. However, if solution quality is determined by the length of time required to perform a given motion, then there is a well defined relationship between time committed to computation and time spent on navigation. When the objective is to have the robot arrive at its goal in the shortest amount of time possible, there will be a threshold after which time invested in computation is not sufficiently rewarded in terms of path improvement.

This optimal computation duration varies greatly for any given environment and start/goal configuration. Additionally, the planner need not decide on a computation duration upfront; the state of the planner and quality of the solution can be observed throughout the process yielding sequential and episodic data. These two facts suggest that deciding when to end a computation phase and begin a navigation phase can be posed as a reinforcement learning problem.

In this work, we present a motion planner that can be trained to minimize the overall time spent on both computation and navigation. To this end, we utilize anytime motion planning techniques as well as reinforcement learning algorithms. The performance of this planner will be evaluated for a variety of simulated environments.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of advisor Professor Stavros Kalafatis and Professor I-Hong Hou of the Department of Electrical and Computer Engineering, and co-advisor Professor Dylan Shell of the Department of Computer Science. All work for the thesis was completed by the student independently.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

Motion planning [1] is the problem of computing feasible and collision free trajectories for a robot from one configuration to another. For a motion to be feasible and collision free, it must respect the physical constraints imposed upon the robot, e.g. maximum acceleration, turning radius, etc., and avoid any overlap of the robot footprint with obstructed space. As input, it takes start and goal configurations and some model or map of the environment. Motion planning is a computationally difficult problem and, in many cases, it may take prohibitively long for an optimal solution to be found. For real applications, it is often desirable for a valid, if sub-optimal, trajectory be returned in a more reasonable amount of time. To address this problem, anytime motion planners have been proposed which can quickly return sub-optimal solutions and iteratively improve upon them.

The use of anytime planners naturally introduces a trade off between computation time and solution quality. Currently, there appears to be no research into this trade off in the context of motion planning. Anytime motion planners [2][3] are typically presented with some arbitrary or empirically determined fixed limit on computation time. However, it will be demonstrated that even the best choice of fixed computation time for a given environment will be outperformed by a planner which behaves ideally. The contents of this thesis are concerned with adaptively predicting a minimizing computation time, specifically in the case in which solution quality is determined by the time required to carry out said solution. In this case, the ideal planner behavior is to minimize the total time spent on both computation and navigation of a solution. Since environment and start/goal poses are varied, the planner must be able to make decisions based on its current internal planning state. This state can be represented in terms of computation time, number of expanded robot configuration states, or any other relevant information. The problem of training the planner to make correct decisions can be formulated as a Markov Decision Process (MDP) [4, chapter 3] where states, actions, and rewards correspond to planner state, incremental increases in computation time, and total time lost or saved over the previous solution. The advantage of posing the problem as an MDP over attempting to predict the best computation time from the outset is that we are able to evaluate the planner state with respect to time and make decisions based on the most recent information.

For the system to behave as envisioned, the motion planner needs to have the following properties: it must be interruptible, meaning computation may be halted at any time while still producing a valid trajectory; it must produce trajectories of non-increasing navigation time with respect to computation time; the trajectories must be parameterized by time; the trajectories are carried out by following the time parameterized path as closely as possible. When these criteria are met, we can plan for some fixed period, evaluate the state of the motion planner, and decide whether to continue computation or begin navigation with the confidence that an estimated trajectory time will be accurate. The steps of a planning episode can then be stated as follows:

1. A pair of robot start and goal configurations are sent to the planner.

2. The motion planner runs until an initial sub-optimal solution is returned.

3. The current planner state and solution are observed by an agent which decides to either continue computation for some fixed increment or begin navigation. If the agent decides to begin navigation, the planning episode ends.

4. The agent receives a reward for having taken a given action in a given planner state.

5. Repeat from (3).

In this work, we leverage reinforcement learning [4] techniques to train such an agent. We present a motion planner meeting the above criteria along with a decision making framework for learning a policy which minimizes combined computation and navigation time. To determine the merit of this approach, we evaluate the performance of the trained agent relative to other fixed computation time policies for three simulated environments.

2. RELATED WORK

2.1 Motion Planning

A motion planning problem consists of an environment, an obstacle region within that environment, a robot model, a configuration space determined by the possible transformations that can be applied to the robot, and a pair of start and goal configurations for the robot [1, p 157]. Motion planning for robotics is a mature field [5, chapter 7] meaning current research is centered around challenging instances of the problem. One such instance is that of planning for differentially constrained robots in continuous space. In this domain, state of the art motion planners typically fall into the categories of sampling based planning and search based planning. Sampling approaches generate random samples from a robots configuration space and assemble these into feasible trajectories. These methods perform well in high degree of freedom problems since they do away with the need to explicitly discretize the configuration space. They also relax the condition of completeness to *probabilistic* completeness, meaning that the probability of returning a solution if one exists approaches one as time increases. Probabilistic Roadmaps [6] sample robot configurations from the entire environment to create a graph, the edges of which are feasible motions computed by a local planner. Once the graph has been completed, multiple queries can be performed using a graph search. Rapidly Exploring Random Trees (RRT) [7] and its anytime extension [3] operate on a similar principle in constructing trees for single searches.

Search based planning is essentially a structured approach to sampling in which sampling occurs on a grid. In a state-lattice, the configuration space is discretized to some resolution and a set of feasible motion primitives connecting neighboring states is computed on- or off-line for each robot configuration. The full set of motion primitives for each configuration is centered on each discrete cell. A grid constructed in such a way consists solely of feasible motions and can be searched efficiently using graph search algorithms. A benefit of this approach is that it permits complex maneuvers, such as three point turns, something absent from algorithms like RRT. With a proper set of motions, every state of the configuration space is reachable and the planner is said to be resolution complete. Pivtoraiko and Kelly [8] propose a principled method for determining a minimal control set of feasible motions for a non-holonomic vehicle, though the control set should ultimately be domain specific. Anytime extensions of this approach have been worked on extensively by the Search-Based Planning Lab [9]. One such algorithm, ARA* [2], uses an inflated heuristic version of A* [10] and allows for the reuse of explored states to increase computational efficiency. The initial heuristic inflation is given as a parameter and iteratively decreased until reaching a value of one, at which point a solution will be optimal up to a given resolution. ADA* [11] extends ARA* to uncertain environments by locally improving segments of the path where changes in cost are observed. Likhachev and Ferguson further introduce the idea of multi-resolution lattices [12] in which a high resolution set of motion primitives is used in the neighborhoods of the robot and the goal and a coarser set is used elsewhere. In this work, the environment is assumed to be static and known *a priori*. As such, ARA* was used as a starting point while incorporating ideas from multi-resolution lattices.

2.2 Reinforcement Learning

Reinforcement learning approaches are concerned with training an agent to take actions within an environment so as to maximize some reward signal. Temporal Difference (TD) learning [4, chapter 6] methods do not require a model for state transitions and are trained using only feedback from the environment. The goal of TD learning is to estimate the expected reward for being in some state s and following some decision policy. In the simplest case, this target is developed recursively with the update rule

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.1}$$

applied after each step, hence called a one-step or TD(0). Here, α is the learning rate, γ is a discounting factor for future rewards, and r_{t+1} is the observed reward. To extend this to the control case, the value being approximated is for a state-action pair. The update rule is then

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{2.2}$$

To address the trade off between exploitation and exploration, actions are selected based on some semi-stochastic policy (*e.g.* ϵ-greedily). In the above update, the action-value function being approximated is for the current decision policy[1]; these approaches are known as on-policy. In the off-policy variant known as Q-Learning, the optimal action-value function is learned, regardless of the current decision policy. This allows for more aggressive exploration with regards to action selection and can help with avoiding local minima at the cost of possibly poor on-line performance. The update for this case is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]. \tag{2.3}$$

In each of the above equations, there is some notion of a value or action-value function to be approximated. The actual representation of this function can take whatever form is convenient. The simplest approach is to use a lookup table wherein the state space takes discrete values along each dimension and each state-action pair is directly assigned a value. In small and discrete domains, a table can be used without loss of information. However, in large and continuous environments, tabular approaches may be ill-suited to the task. They require that each cell be updated many times before the value approximation converges, and they lack the ability to reuse information for similar states. On top of this, the size of the table grows exponentially with the dimension and quantization of the state-action space.

To extend TD methods to continuous state spaces, the approximate value function can be parameterized by a weight vector **w** so that $Q(s, a) \approx Q(s, a, \mathbf{w})$. A semi-gradient descent update

[1] A canonical example of the difference between on-policy and off-policy methods is cliff walking. The on-policy agent will learn to avoid the edge of the cliff due to the possibility of randomly walking off the edge while the off-policy agent will learn to take the most direct path along the edge of the cliff.

for this **w** in the on-policy control case takes the form

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}, \mathbf{w}_t) - Q(s_t, a_t, \mathbf{w}_t)]\nabla Q(s_t, a_t, \mathbf{w}_t), \tag{2.4}$$

with a modification analogous to that shown previously for off-policy control. Linear function approximation offers some of the strongest guarantees on convergence and stability [4, p 166], but it also requires appropriate feature selection. Some successfully demonstrated feature choices for linear function approximation include linear weighting, polynomial bases, Fourier-bases [13], and Radial Basis Functions [14]. Non-linear methods lack theoretical guarantees on stability or convergence but offer some of the most compelling successes in the reinforcement learning literature, surpassing the best human performance in the domains of Go [15] and Atari games [16]. In both of these examples, artificial neural networks were used in conjunction with some variant on value function approximation.

In this work, we present results using both tabular and neural net function approximators. Using a table provides a baseline for what we can expect from the simplest kind of approach. From there, we can better evaluate the benefit of increased complexity in the neural net case.

3. PLANNER DESIGN

The planner presented here consists of an anytime motion planner and an integrated decision making agent. The terms motion planning and trajectory planning are often used interchangeably. For clarity, the motion planner will hereafter be referred to as a trajectory planner. The combined trajectory planner and learning agent will be referred to simply as the planner. As previously stated, we desire for the trajectory planner to exhibit the following properties:

1. computation is interruptible,
2. trajectories are parameterized by time,
3. solutions are trajectories of non-increasing navigation time with respect to computation time.

The first property allows the state of the trajectory planner to be evaluated with respect to a reward signal at regular intervals. The second property allows the agent to effectively estimate the navigation time for a trajectory. This ability is required for any reward signal to be designed. The third property ensures that the trajectory planner solution quality is aligned with the desired agent behavior. While this property may not be strictly necessary to train an agent, we strongly suspect that deviation from it would negatively impact trainability. To enable the latter two properties, we must first design an appropriate set of motion primitives.

3.1 Motion Primitives

The robot model we use is a non-holonomic wheeled robot which can move forwards and backwards. We use a four dimensional state representation (x, y, θ, v) where x, y represent the position of the center of the robot, θ is the orientation, and v is velocity. For this application, we use 16 possible orientations, and 7 velocities. Each motion primitive represents a path from $(0, 0, \theta_i, v_i)$ at time $t_i = 0$ to $(x_f, y_f, \theta_f, v_f)$ at t_f where (x_f, y_f) snap to a grid of a given resolution. For each starting configuration, a set of neighboring configurations are computed for which paths fall within some limits on velocity and acceleration. As the velocity increases, we increase the size of the neighborhood. Next, we generate intermediate x, y, θ, t values that act as time-stamped waypoints along the path. In this way, motion primitives can be concatenated to generate a time parameterized path with a granularity corresponding to the number of intermediate points. Motion primitives are calculated off-line and stored in a lookup table for use during planning.

This approach differs from the multi-resolution lattice structures presented in [12] in that we are explicitly accounting for velocity. If we were to only use some nominal forward and backward velocities as they do, we would only be able to loosely reason about the time required to carry out trajectories. We are also introducing a somewhat modified notion of multi-resolution. Rather than using a high resolution lattice near the goal and a coarse subset elsewhere, the resolution of our lattice decreases as velocity increases. This reflects the reality that when a higher lattice resolution is needed, such as near a goal or while maneuvering through obstacles, the robot will be travelling slowly. Figure 3.1 shows a subset of motion primitives for one initial heading.

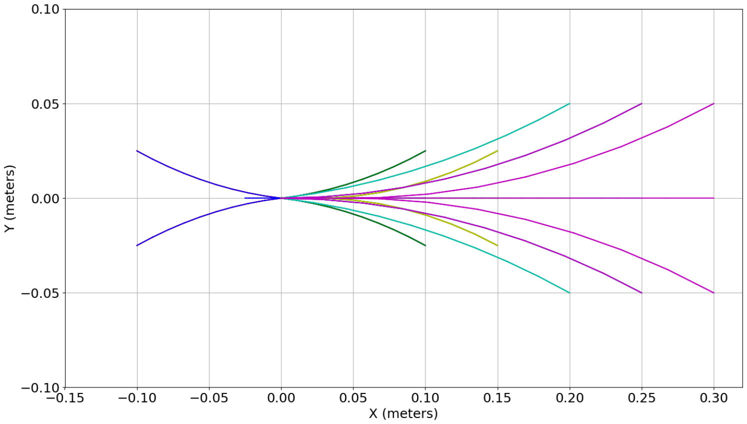


Figure 3.1: Subset of motion primitives for an initial heading of 0°. Different colors correspond to different initial velocities.

3.2 Trajectory Planner

3.2.1 Search Algorithm

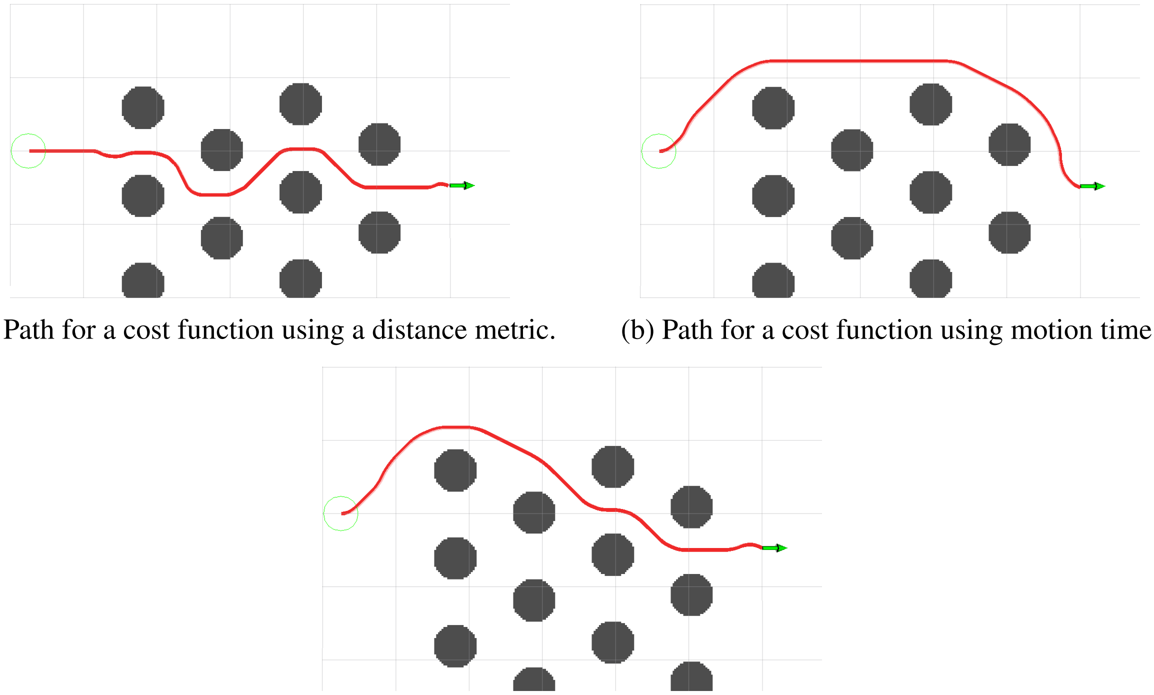
To convert motion primitives into a path, we use ARA*, a heuristic-based graph search algorithm developed by Likhachev, Gordan, and Thrun [2]. We limit our discussion here to points which will be relevant in the following sections. Regular A* maintains a priority queue (denoted *open*) sorted by $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of reaching s from s_{start} and heuristic $h(s)$ represents the estimated cost of reaching s_{goal} from s. For an optimal solution to be guaranteed, $h(s)$ needs to be consistent, *i.e.* $h(s) \leq c(s, s') + h(s')$ for any successor state s' of $s \neq s_{goal}$ and $h(s_{goal}) = 0$. Here, $c(s, s')$ is the stictly positive cost of an edge going from s to s'. This means that $h(s)$ never overestimates the cost of reaching s_{goal} from s. Inflating $h(s)$ by a factor of $\epsilon > 1$, violates this property, but in doing so results in fewer states being expanded and thus faster computation. A crude anytime algorithm could be constructed by executing ϵ inflated A* successively while decreasing ϵ, but this would be inefficient as it does not reuse information from one iteration to the next.

ARA* is able to reuse the results of previous searches by introducing a concept of local inconsistency, wherein a state s is locally inconsistent from when its cost $g(s)$ is decreased up until it is revisited. The state s is locally inconsistent in the sense that while $g(s)$ has been updated, any successor states s' of s remain unchanged. Once s is expanded, this inconsistency is corrected and propagated to the successors of s'. Once a state is made inconsistent, it is inserted into the *open* list. Each state is restricted to being inserted into the *open* list only once. However, an ϵ inflated heuristic violates consistency meaning that a state may be re-expanded multiple times. To account for this, an additional *incons* list is maintained of all locally inconsistent states which have been previously expanded. After each planning iteration and ϵ decrease, a new *open* list is built from the union of *open* and *incons* with updated $f(s) = g(s) + \epsilon \cdot h(s)$ values.

One further notion is that of a sub-optimality bound ϵ'. The authors calculate this in two ways. First, after each search iteration the solution returned will be sub-optimal by at most a factor of ϵ. In the second way sub-optimality is computed as the ratio between $g(s_{goal})$ and the smallest unweighted $f(s)$ value of an inconsistent state. The cost of a solution $g(s_{goal})$ provides an upper bound since this cost can be only be improved upon, and $f(s)$, being the most optimistic estimation of solution cost, provides a lower bound. By taking the minimum of the two approaches as in Equation 3.1, the minimum bound is found:

$$\epsilon' = \min[\epsilon, g(s_{goal})/\min_{s \in open \cup incons}(g(s) + h(s))]. \tag{3.1}$$

3.2.2 Cost Function



(a) Path for a cost function using a distance metric.

(b) Path for a cost function using motion time.

(c) Path for a cost function using both time and distance.

Figure 3.2: Path behavior for different cost functions.

A search algorithm like A* minimizes $g(s_{goal})$, the cost of reaching the goal by way of edges on a graph. In the context of this work, these edges are motion primitives and the quantity to be minimized is the time required for the robot to carry out a solution. This presents a more challenging problem than simply minimizing some distance metric. For example, consider the hypothetical situation in Figure 3.2. While the shortest route to the goal goes through the obstacle region, its constrained nature means that the robot must move slower. A quicker option might be to circumvent the obstacle region altogether. A first attempt at encoding this information into the edge costs might be to use only the time required to perform the motion primitive. However, in that case, short, low speed motions may be evaluated the same as longer high speed motions. To encourage decreasing navigation times, distance and time must both be considered as in Equation 3.2:

$$c(s, s') = w_d \cdot d(s, s') + w_t \cdot \Delta t(s, s'), \tag{3.2}$$

where $d(\cdot, \cdot)$ is a distance metric, $\Delta t(\cdot, \cdot)$ is the time required for a motion, and w_d, w_t are weights. The weights can be tuned to achieve the desired behavior. Experimentally, we found that a $w_d : w_t$ ratio of $1 : 10$ was best for our motion primitive set, but different acceleration and velocity limits may require that those weights be adjusted. Equation 3.2 does not result in strictly decreasing navigation times, but it performs well in practice: in an experiment of 100 planning episodes consisting of 5420 search iterations, 297 of these iterations resulted in increased navigation time with a mean increase and standard deviation of $0.656\,\mathrm{s}$ and $0.464\,\mathrm{s}$ respectively. These increases are generally corrected in subsequent searches, and we can further restrict the published solutions to only those which decrease navigation time. In other words, while learning and when passing solutions along for navigation, we discard any changes to a solution which result in an increase in the projected navigation time. An illustrative set of these times are shown in Figure 3.3, where each color corresponds to a separate start/goal configuration computation sequence. We can see the general relationship between computation time and projected navigation time, including occasional increases in the latter.

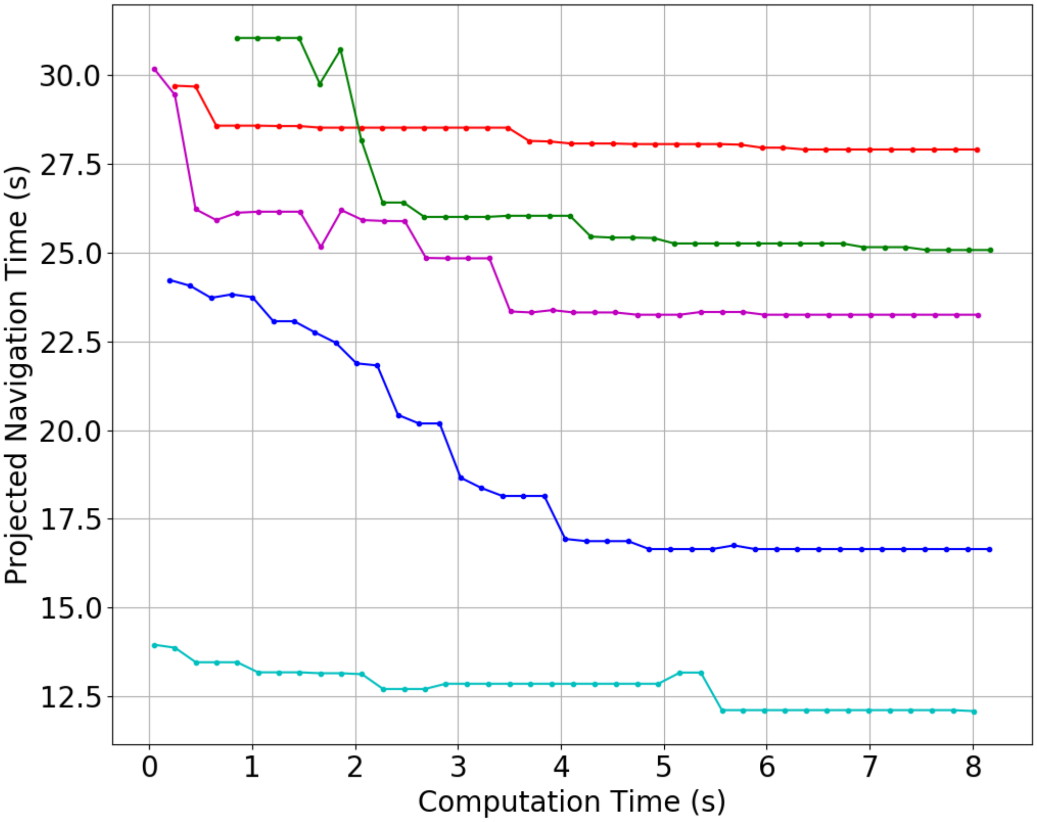


Figure 3.3: Five sampled planning episodes illustrating the relationship between computation time and projected navigation time.

3.3 Learning Framework

In order to use reinforcement learning techniques, we require some notion of a reward or feedback signal. Consider the following steps of a planning episode:

$$\begin{aligned}
t_0^A &= t_0^C + t_0^N \\
t_1^A &= t_0^C + t_1^C + t_1^N \\
t_2^A &= t_0^C + t_1^C + t_2^C + t_2^N \\
&\vdots \\
t_n^A &= t_0^C + t_1^C + ... + t_n^C + t_n^N
\end{aligned} \tag{3.3}$$

where t_i^C is the amount of time spent on computation in step i, and t_i^N is a reliable projected navigation time at step i, and t_i^A is the projected time of arrival at the goal. We have no *a priori* knowledge of whether a t_i^N is good or bad, but we do have t_0^N as a reference point. Consequently, the quantity that we want to minimize is the total time lost or saved over the initial t_0^A plan. At each step, this can be expressed as:

$$\begin{aligned}
r_i &= \begin{cases} t_{i-1}^A - t_i^A & \text{for } i \geq 1 \\ 0 & \text{for } i = 0, i = n \end{cases} \\
&= \begin{cases} t_{i-1}^N - t_i^C - t_i^N & \text{for } i \geq 1 \\ 0 & \text{for } i = 0, i = n. \end{cases}
\end{aligned} \tag{3.4}$$

An episodic reward characterized by $r = \sum_{i=0}^{n} r_i$ will yield the projected time lost or saved over the entire episode with respect to t_0. If r is positive, then $t_n < t_0$ meaning that time has been saved. Similarly, if r is negative, then $t_n > t_0$ and time has been lost. The objective of the agent is then to learn a step-wise decision policy which minimizes r. This can be framed as Q-learning problem with the modified update rule given by Equation 2.3, where the learned action value function $Q(s, a)$ approximates the expected reward, with a discounting rate of γ, for following the learned policy in state s. Once trained, the optimal policy will be to choose action a as $\arg\max_a(Q(s,a))$. The action set here consists of two choices: the agent can compute for a fixed increment of $0.2\,\mathrm{s}$, or it can begin navigation. The choice of navigation results in a terminal state, meaning that the reward is zero. In this way, once the expected future reward becomes negative, meaning that no further improvement over the initial plan is expected, the most appealing action will be to begin navigation.[1] This has the added benefit that we need only learn the value function approximation for a single action, that of further computation, since the estimated value of a terminal state with a reward of zero will remain zero.

3.3.1 Tabular Q-Learning

To represent $Q(s,a)$, we can use a lookup table. A table requires that both the state and action spaces be quantized. For actions, this is trivial as the action set has a cardinality of 2. For the state space, more work is required. As discussed in Chapter 1, our concept of state is limited to the information internally available to the planner. This includes nine basic attributes:

- current time relative to receiving goal,
- path navigation time and cost,
- initial heuristic value to goal,
- current heuristic inflation,
- sub-optimality bound,
- list sizes, *e.g. open*, *incons*, and *closed*.

We consider these quantities as dimensions of the planner state and then divide dimensions up into bins. Here, we use quantile binning on precollected data to determine appropriate boundaries. In this way, assuming a fixed number of bins k, the state space is represented by k^d discrete features

[1] In the case of $Q(s,a) = 0$, an agent would be indifferent between its two choices of action.

where d is the dimensionality. Since there is only a single action on which learning occurs, this is also the size of the lookup table. For future notational consistency, we can consider $Q(s, a)$ as

$$Q(s, a, \mathbf{w}) = \mathbf{1}(s, a) \cdot \mathbf{w}, \tag{3.5}$$

an indicator function with a value of 1 if (s, a) is present and weights corresponding to table entries.

In constructing the table, there are two main issues to balance: information loss due to binning and trainability. Constraints on system memory are not an issue since before that limit is ever approached, the agent would be effectively untrainable. This is because, as alluded to in Chapter 2, a table does not generalize information across states and states must be visited many times before accurate estimates can be obtained. Even if a set of states has been traversed frequently enough for a stable value to emerge, if a nearly identical neighboring state is seen for the first time, none of the previous experience will be applicable. If data were cheaply obtained, we could tolerate a table with perhaps many tens or even hundreds of thousands of entries. However, the data we're dealing with take several seconds to generate for each planning episode. Thus, for convergence in training, a smaller table is better. The disadvantage is that less information is represented by a smaller table. For example, the size of the *open* list can in practice range from a few hundred to a few hundred thousand. Using thousands of bins will be intractable for obvious reasons, but a more reasonable number like $k = 5$ will result in large ranges of values being indistinguishable. Likewise, the number of dimensions must be restricted. These limitations are inherent in a tabular approach when the state space is large and continuous. For the results of this work, values of $k = 5$ and $d = 6$ were used for a tractable table size of 15625.

The choice of planner attributes used in constructing features relies on experimentation and intuition. The six quantities that were settled on are: total computation time, the sub-optimality bound on the solution, *open* and *incons* list sizes, and the time derivatives of *open* and *incons* list sizes. The usefulness of total computation time makes sense as all other quantities are dependent on this. Navigation time would seem to be of value, but we found the sub-optimality bound to be more informative. The intuitive argument in favor of using navigation time is that a longer path might have more room for improvement. However, we can imagine two hypothetical cases: the first has a navigation time of, say, five seconds with a large sub-optimality bound and the second has an navigation time of twenty seconds with a bound close to unity. In the case of the shorter path, a few seconds of improvement might be attained whereas the longer path is already nearly optimal. With regards to the lists, in initial experiments we used only their sizes with unimpressive results. We then experimented with the differences in list sizes from step to step as this indicates the internal activity of the planner. Using list rates of change provided marginally better though still unsatisfying results. By using both list sizes and rates of change, *i.e* location and activity along those dimensions, we obtained superior results to those of either feature set on its own. Still, we expect better results to be achievable if we can afford to be less discriminating in terms of the information provided to the learning agent. In the next section, we present an approach that enables us to use all of the planner attributes.

3.3.2 Q-Learning with Artificial Neural Network

Representing $Q(s, a, \mathbf{w})$ with a neural network addresses the limitations of a table and takes care of feature extraction automatically. We used a fully connected architecture with two hidden layers of size 10 and a $\tanh$ activation function. The input layer is of size twelve: the nine attributes described previously which comprise the planner state s along with the time derivatives of the three lists. The network has a single output corresponding to the expected future reward for taking the action of incremental computation while in state s and thereafter following the learned policy. Clearly, the inputs of the network are on vastly different scales, so each attribute is transformed to have zero-mean and unit-variance.[2] For an $\langle s, a, r, s' \rangle$ sequence, the target for the neural network is

$$r + \gamma \arg\max_{a'} Q(s', a', \mathbf{w}) \tag{3.6}$$

[2] In training the network, data were pre-collected so that means and variances were easily computed. In the spirit of reinforcement learning, we experimented with an on-line single pass method for calculating mean and variance during training using Welford's algorithm [17, p 232] and achieved comparable results.

and we use an L2 loss function, *i.e.*

$$\frac{1}{2}(r + \gamma \text{arg}\max_{a'} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}))^2. \tag{3.7}$$

As suggested by Mnih *et. al* [16], we made use of experience replay in training the network. Experience replay refers to the concept of storing $\langle s, a, r, s' \rangle$ sequences in memory as experiences for future use. In the course of training, the agent will immediately learn from such a sequence, store the experience, and then randomly sample some number of experiences from its memory to train on as well. This makes for a more efficient use of data as experiences are reused possibly many times rather than being discarded. More importantly, it also de-correlates the observed input sequences. In the naive approach, the network will see a sequence of similar inputs (a planning episode) and adjust weights accordingly. Upon the next planning episode, weights will be adjusted in a different direction and so on. By splicing in experiences randomly, we create what amounts to de-correlated mini-batches of data which more closely resemble the *i.i.d.* data assumed in neural network back-propagation algorithms.

With the trajectory planner and learning framework in hand, we can now present the full algorithm in Figure 3.4. Since we are dealing with two different concepts of state, lowercase s will refer to the (x, y, θ, v) configuration space of the robot while uppercase S will denote the planner state. This algorithm is general for off-policy approaches. The differences between the tabular and neural net approaches are encapsulated in the update on line 22.

```
procedure MAIN()
    initialize weights $\mathbf{w}$
    initialize heuristic inflation $\epsilon$
    initialize policy greediness $\eta$
    initialize open, incons, closed to $\emptyset$
    $g(s_{goal}) \leftarrow \infty$
    $g(s_{start}) \leftarrow 0$
    insert $s_{start}$ into open with fvalue($s_{start}$)
    ImprovePath($time\_limit$)
    $S \leftarrow$ planner state
    $A \leftarrow \pi(S, \mathbf{w}, \eta)$
    while $\epsilon > 1$ and $A \neq 0$ do
        $time\_limit \leftarrow$ now() $+ A$
        while now() $< time\_limit$ and $\epsilon > 1$ do
            if ImprovePath not timed out then
                decrease $\epsilon$
                move states from incons to incons
                update priorities $\forall s \in$ open by fvalue($s$)
            closed $\leftarrow \emptyset$
            ImprovePath($time\_limit$)
        $R \leftarrow$ reward($S, A$)
        $S' \leftarrow$ planner state
        update $Q(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow \pi(S, \mathbf{w}, \eta)$
    publish current solution
```

```
procedure $\pi(S, \mathbf{w}, \eta)$
    $A \leftarrow \arg\max_{A'}(Q(S, A', \mathbf{w}))$with probability $\eta$
    otherwise $A \leftarrow$ planning increment of $0.2$
    return $A$
```

```
procedure FVALUE($s$)
    return $g(s) + \epsilon \cdot h(s)$
```

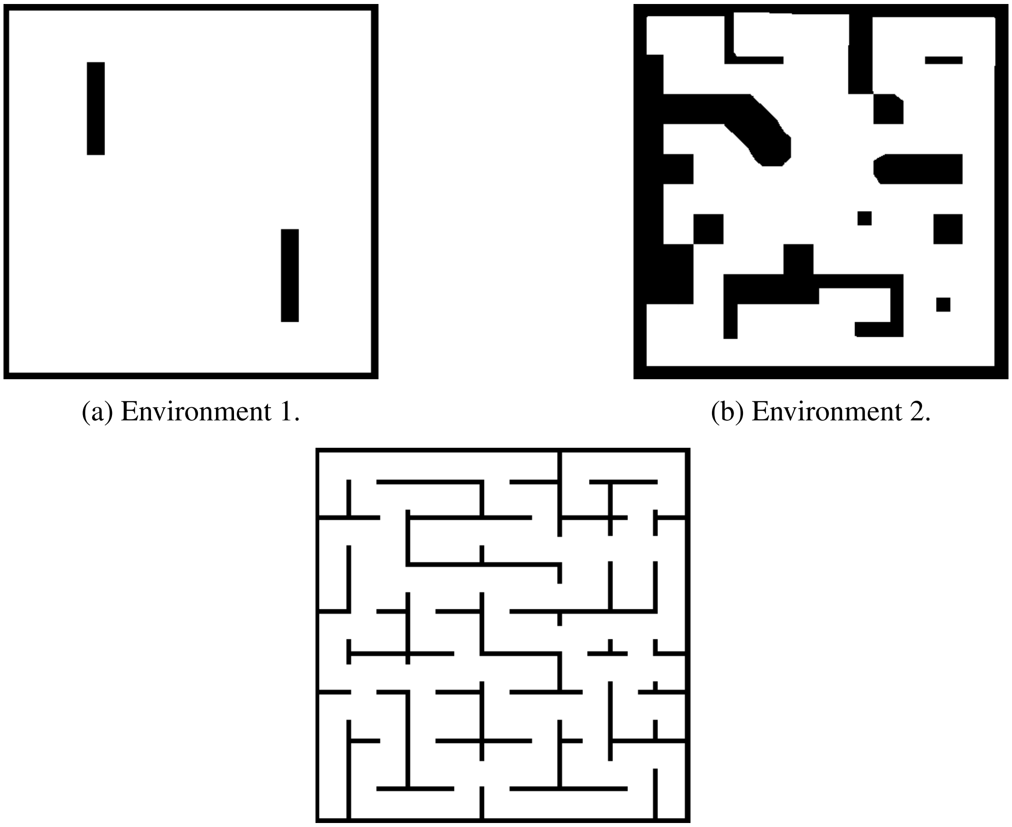
```
procedure IMPROVEPATH($time\_limit$)
    while fvalue($s_{goal}$) $> \min_{s \in open}$ fvalue($s$) and now() < $time\_limit$ do
        pop $s$ from open
        closed $\leftarrow$ closed $\cup \{s\}$
        for each successor $s'$ of $s$ do
            if $s'$ not visited then
                $g(s') \leftarrow \infty$
            if $g(s') > g(s) + c(s, s')$ then
                $g(s') \leftarrow g(s) + c(s, s')$
                if $s \notin$ closed then
                    insert $s'$ into open with fvalue($s'$)
                else
                    insert $s'$ into incons
```

Figure 3.4: Combined trajectory planner and learning framework.

4. METHODOLOGY AND DISCUSSION

The planner is simulated on a Roomba-like differential drive robot with a maximum forward velocity of $0.5\,\mathrm{m/s}$. We use a motion primitive set with 16 possible orientations, 7 velocities values, and 56 primitives per heading for a total size of 896. To speed computation, a fast 8-connected 2D (x, y) Dijkstra search is used as the heuristic for the 4D (x, y, θ, v)-lattice trajectory planner. The initial heuristic inflation value is set to 4.0 and decremented in steps of 0.2. We use three environments of differing characteristics, each of which is $10\,\mathrm{m} \times 10\,\mathrm{m}$ and discretized to a resolution of $2.5\,\mathrm{cm}$ for a search space of approximately 18 million configurations. The first environment is largely devoid of obstacles, the second is moderately cluttered with obstacles, and the third is highly constrained with narrow corridors. The three environments are depicted in Figure 4.1.

For training data, we sample 6000 start/goal configurations randomly from each environment where each pair is ensured to admit a feasible solution. For each sample, the planner is allowed to run for $10\,\mathrm{s}$ and the planner state is recorded at intervals of $0.2\,\mathrm{s}$, corresponding to an action. We choose $10\,\mathrm{s}$ as a cutoff as, after that, improvements in the overall time spent are exceedingly rare. For validation and testing data, we collect sets of 500 configuration pairs in a similar manner. During training, the planner is evaluated on validation sets at regular intervals in order to monitor convergence. Once a model is trained, it is assessed on a test set. To provide a frame of reference, we process the validation and test sets to determine the minimizing computation time for each configuration pair, thereby attaining a lower bound on what is achievable. In the following figures and tables this bound is referred to as the optimal or ideal behavior of a planner. Note that this does not refer to optimality in the sense of an optimal solution as returned by the *trajectory* planner; rather, it is optimal from the perspective of an agent trying to minimize the total time spent on both computation and navigation. We are also interested in how a trained planner performs relative to a naive approach in which a fixed amount of additional computation time is allotted once an initial solution has been found. As such, we also find the minimizing *fixed* computation time over each



(a) Environment 1.

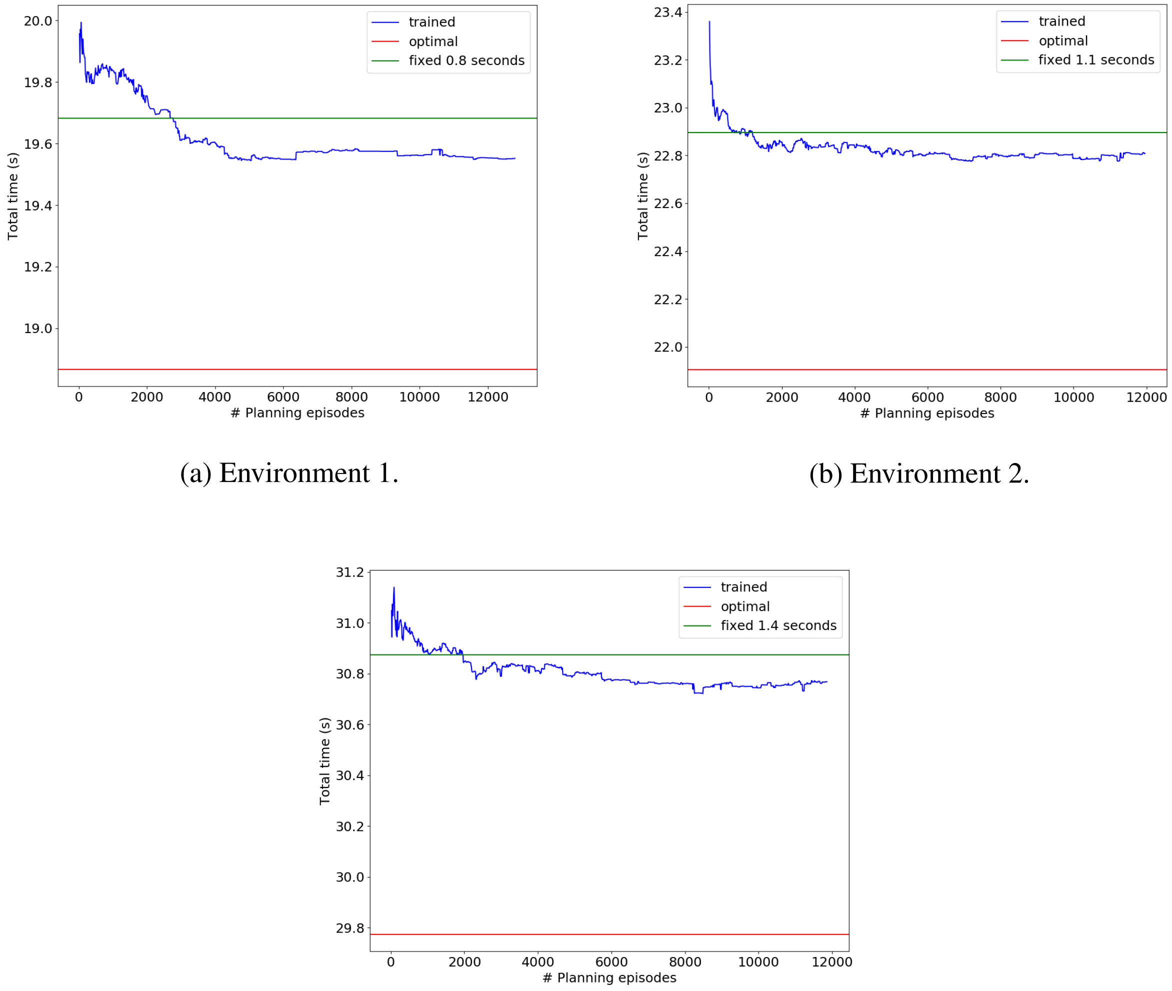
(b) Environment 2.

(c) Environment 3.

Figure 4.1: Simulated environments.

training set and use this to determine the best fixed policy performance on testing and validation sets. In all cases, the timing information presented here is being averaged over an entire set.

For experiments with a tabular approach to value function approximation, we use an initial learning rate of $\alpha = 0.01$ and a discount factor of $\gamma = 0.95$. We leverage an off-policy approach by aggressively exploring during training. That is, the policy being followed is to always incrementally compute up until the $10\,\mathrm{s}$ cutoff is reached. For the plots shown in Figure 4.2, the training sets were passed through twice with the learning rate halved after the first iteration. We observed that at the end of a full pass through the training sets, previously unseen states were still encountered with some frequency, indicating that our table of size 15625 may be nearing the limit of trainability given the quantity of training data.

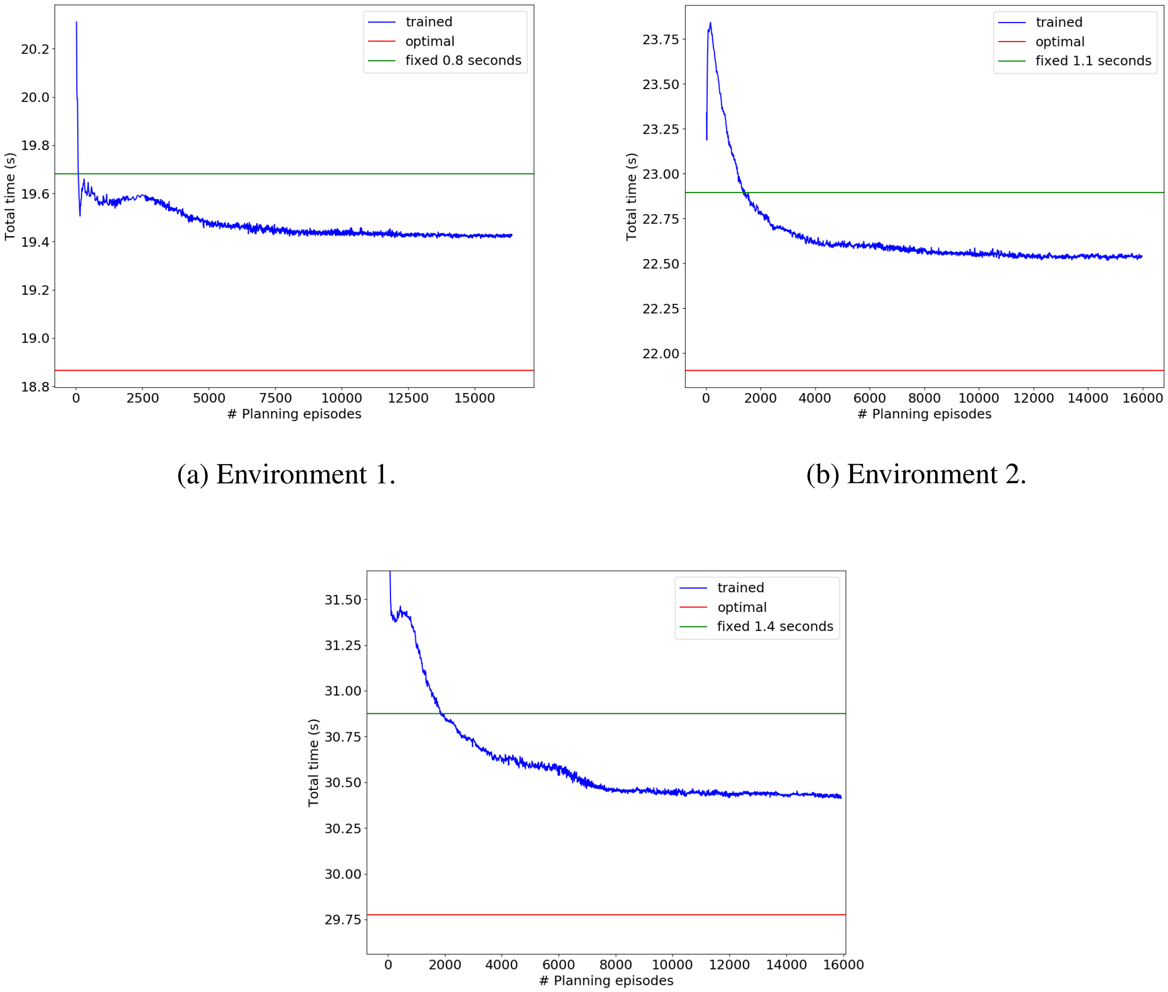


(a) Environment 1.

(b) Environment 2.

(c) Environment 3.

Figure 4.2: Tabular training results for each environment.



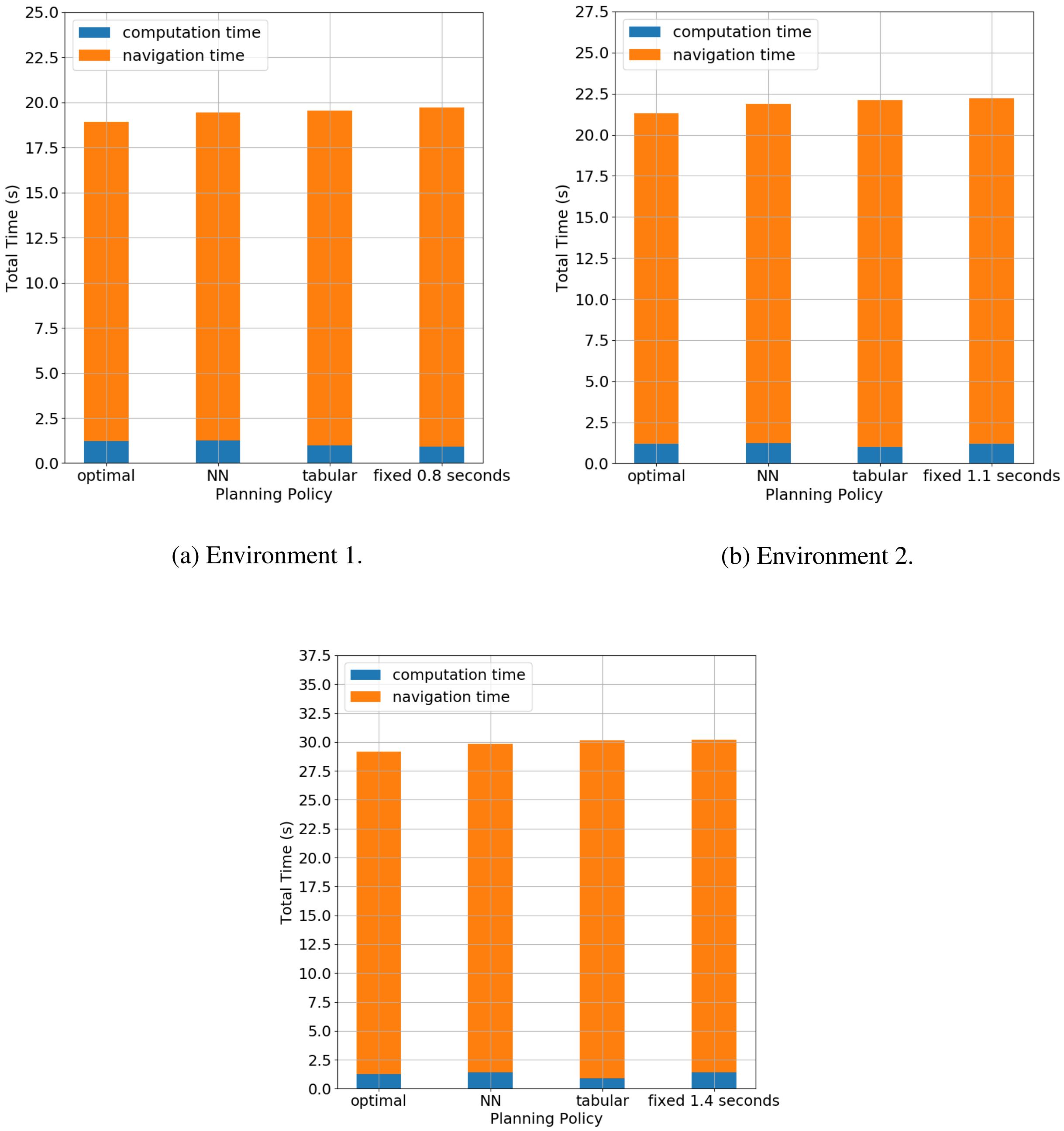
(a) Environment 1.

(b) Environment 2.

(c) Environment 3.

Figure 4.3: Neural network training results for each environment.

For the neural network approach, we use an initial learning rate of $\alpha = 0.0002$ and a discount factor of $\gamma = 0.95$. After several hundred initial experiences, the replay memory is sampled 50 times for each planning step. Again, we explore aggressively during training. Once all training data have been seen, the learning rate is halved and the network is further trained using the built up replay memory for a rough equivalent of 6,000 planning episodes. This step is performed twice. Training progress is shown in Figure 4.3. For both training methods, performance is evaluated on validation data at intervals of 20 planning episodes, translating to roughly 800 $\langle s, a, r, s' \rangle$ experiences. The performances of the trained planners on the testing sets are summarized in Figure 4.4 and Table 4.1.



(a) Environment 1.

(b) Environment 2.

(c) Environment 3.

Figure 4.4: Comparison of trained planners relative to optimal and fixed planning policies.

Policy	Computation time (s)	Navigation time (s)	Total time (s)
Optimal	1.236	17.678	18.913
NN	1.264	18.190	19.454
Tabular	0.967	18.587	19.554
0.8 seconds	0.918	18.805	19.722

(a) Environment 1.

Policy	Computation time (s)	Navigation time (s)	Total time (s)
Optimal	1.178	20.138	21.316
NN	1.223	20.655	21.878
Tabular	1.012	21.087	22.099
1.1 seconds	1.190	21.031	22.220

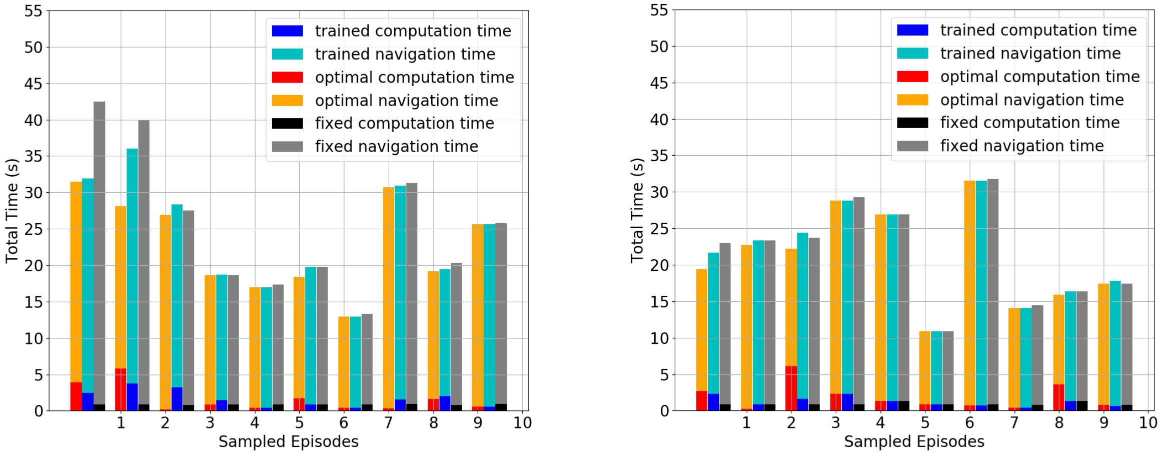
(b) Environment 2.

Policy	Computation time (s)	Navigation time (s)	Total time (s)
Optimal	1.259	27.912	29.171
NN	1.423	28.437	29.860
Tabular	0.917	29.255	30.172
1.4 seconds	1.441	28.782	30.223

(c) Environment 3.

Table 4.1: Time comparison of trained planners relative to optimal and fixed planning policies.

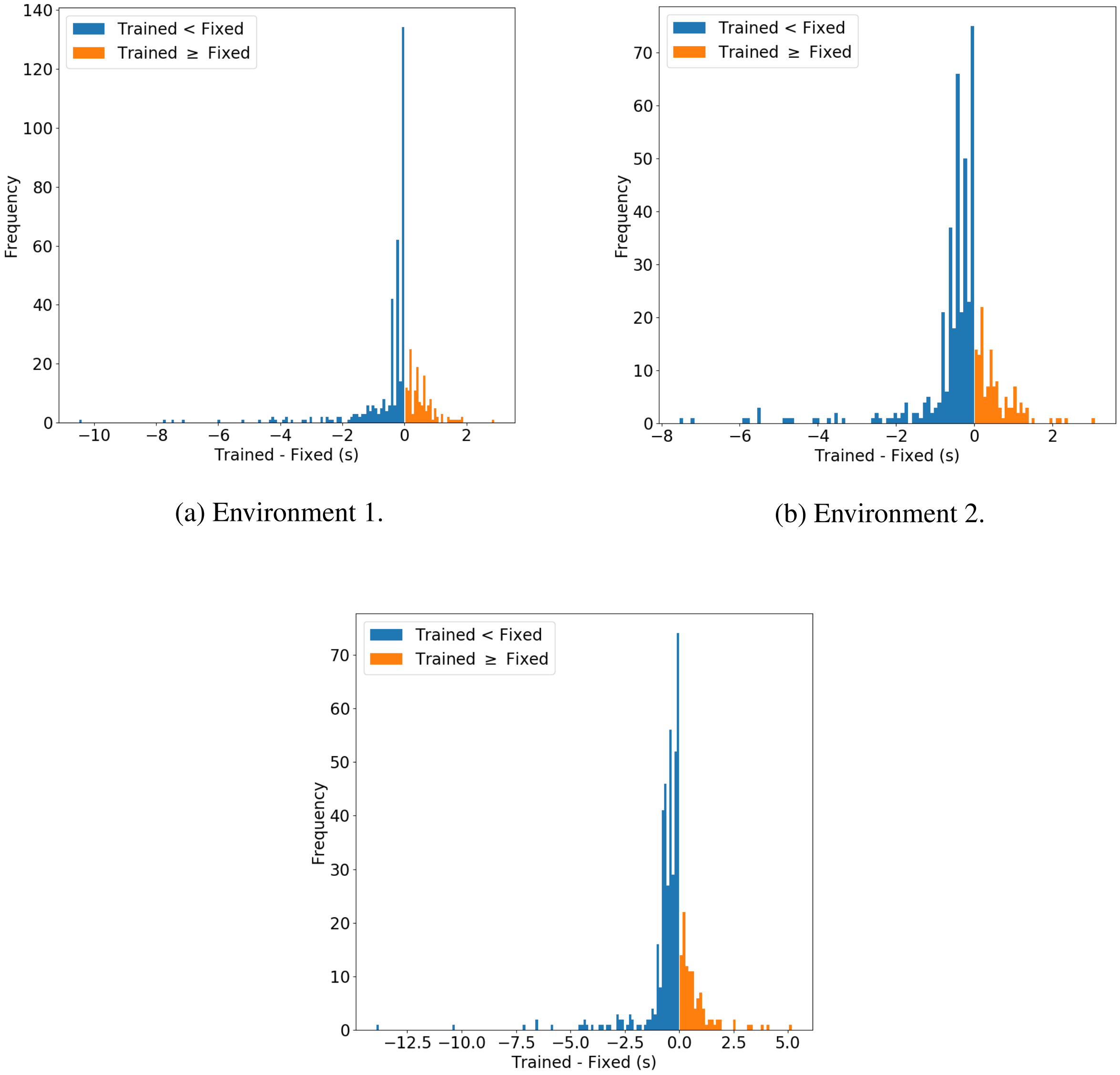
In terms of the difference between the optimal behavior and the learned behavior, the planner appears to perform quite well. It is when we compare these results to a best case fixed time policy that we see the margin for improvement is actually quite small at roughly $1\,\mathrm{s}$ for all simulated environments. That said, both training methods were able to outperform the best fixed policy for each of the environments tested. As expected, neural nets fared better than a table in all cases, but they miss the mark of the optimal policy by around $0.6\,\mathrm{s}$ for each of the environments. Since these metrics used are averaged over many planning episodes, they fail to illustrate what is happening on a per-episode basis. To help visualize this, we've plotted randomly selected sampled episode timings for the neural network approach in Figure 4.5. What we see is that the trained behavior closely follows the optimal behavior except in cases where the ideal computation time is relatively large. We would expect that the planner fails to predict for such outlying cases, instead favoring shorter and safer computation times. We can quantify this in some sense by observing that, for example, in the case of the sparse environment, the standard deviations of the optimal and learned computation times are $1.37\,\mathrm{s}$ and $0.59\,\mathrm{s}$ respectively. In fact, for the sparse environment test set, the longest computation time for the trained planner was $4.125\,\mathrm{s}$ (and in that particular case, the first path improvement would have occurred at $8.8\,\mathrm{s}$ of planning). To determine the effect of such outliers on average performance, we evaluated the planner on test sets with all samples having an optimal computation time greater than $5\,\mathrm{s}$ removed. We found that disparities in overall time taken remained around the same at $0.6\,\mathrm{s}$, indicating that it is the accumulation of small mispredictions rather than dramatic outliers that dominate these results.



(a) Environment 1.

(b) Environment 2.

Figure 4.5: Timing comparison for several randomly selected episodes.



(a) Environment 1.

(b) Environment 2.

(c) Environment 3.

Figure 4.6: Total time lost or saved by trained policies relative to best fixed policies, evaluated for all episodes of testing sets.

To reiterate, we don't expect the trained planner to exactly match the behavior of an idealized perfect decision maker. A more attainable goal is for the trained planner to outperform a best fixed policy. We've seen previously that for a sample size of 500, we indeed see an average improvement. However, we would like to reason about what we can expect for a smaller sample size. In other words, how confidently can we say that a trained planner will perform better than a fixed policy on a given start/goal pair. To determine this we can look at the difference between the total times spent by trained and fixed policies for each sample in a test set; when the difference is negative, the trained policy has saved time. We find that in nearly 80% of samples, the trained policy saves time over a fixed policy. Moreover, the number of instances of large time savings (say, two times the standard deviation of the differences over a set) greatly outnumber cases where a large amount of time is lost. These findings are visualized in Figure 4.6 and summarized in Table 4.2.

Case	Number of instances	Mean difference (s)	Std. deviation of difference
Trained $<$ Fixed	370	-0.615	0.765
Trained $\geq$ Fixed	130	0.395	0.374

(a) Environment 1.

Case	Number of instances	Mean difference (s)	Std. deviation of difference
Trained $<$ Fixed	397	-0.592	0.759
Trained $\geq$ Fixed	103	0.651	0.634

(b) Environment 2.

Case	Number of instances	Mean difference (s)	Std. deviation of difference
Trained $<$ Fixed	392	-0.718	1.263
Trained $\geq$ Fixed	108	0.757	0.878

(c) Environment 3.

Table 4.2: Summary of data shown in 4.6.

One final experiment of interest is how a generalist planner trained on a variety of environments might perform. For this, we only look at the neural network approach. We set all parameters as before, but in this case, we train and evaluate using the datasets for each of the environments. We again compare trained performance to the optimal average and the best average fixed time policy over all training data. The training progression for this is shown in Figure 4.7. The planner is evaluated on the combined test sets for all environments as well as each test set individually. These results are summarized in Table 4.3. They show that a neural network cross-trained on multiple environments performs nearly equally as well as one trained on a specific environment. In the aggregate, these experiments show that an adaptive planner can yield a small but consistent average performance improvement over a static fixed time approach.

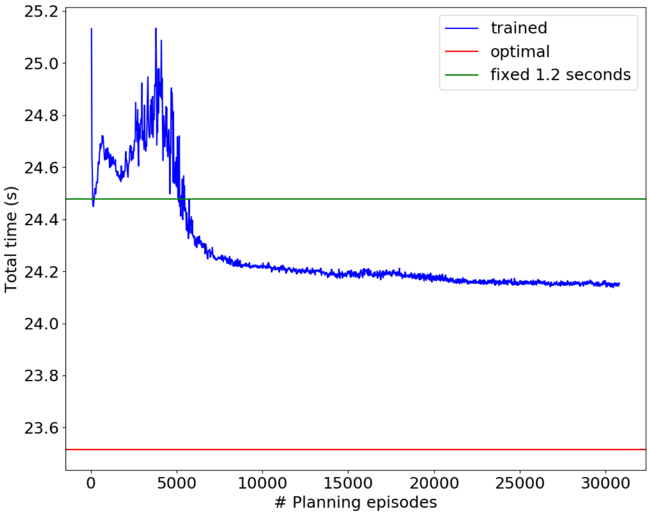


Figure 4.7: Neural network trained on all environments.

Policy	Computation time (s)	Navigation time (s)	Total time (s)
Optimal	1.224	21.909	23.133
NN	1.309	22.437	23.747
1.2 seconds	1.208	22.858	24.065

(a) All environments.

Policy	Computation time (s)	Navigation time (s)	Total time (s)
Optimal	1.236	17.678	18.913
NN	1.140	18.309	19.449
1.2 seconds	1.28	18.472	19.753

(b) Environment 1.

Policy	Computation time (s)	Navigation time (s)	Total time (s)
Optimal	1.178	20.138	21.316
NN	1.169	20.744	21.910
1.2 seconds	1.273	20.996	22.269

(c) Environment 2.

Policy	Computation time (s)	Navigation time (s)	Total time (s)
Optimal	1.259	27.912	29.171
NN	1.627	28.334	29.961
1.2 seconds	1.257	28.975	30.232

(d) Environment 3.

Table 4.3: Time comparison of generalist trained planner relative to optimal and fixed planning policies.

We've seen that for the environments studied here, there is not much room for improvement over a best fixed policy and that outlying cases make matching an optimal decision maker difficult. We could possibly see some marginal performance gain by reducing the incremental computation time from $0.2\,\mathrm{s}$ down to, say, $0.1\,\mathrm{s}$, but it is doubtful that this would yield much more than the difference we would see due to stochasticity in weight initialization and experience replay sampling. A more plausible avenue for improvement lies in the discussion of Section 3.2.2. We saw that navigation times, while generally decreasing, are not monotonically decreasing. We work around this by only publishing plans which improve navigation time, but this behavior is not intrinsic to the planner. When we discussed the intuition behind feature choices like list rates of change, we saw them as describing activity or lack thereof along certain dimensions. Since the solution quality as defined by our cost function is not strictly aligned with our learning objective of minimizing total time, such features can indicate an impending change in the solution which will in fact be suppressed due to an increased navigation time. Internally, this looks no different to the learning agent than a decrease in navigation time, but it will not receive a corresponding reward. These confounding cases clearly make learning more difficult, but the frequency of their occurence remains low enough to be able to effectively train on varied environments.

5. FUTURE WORK

In this thesis, we have constrained the problem in ways that might be undesirable for a system deployed in the real world. Namely, we have assumed that the possible actions consist only of incremental computation and navigation, and that the environment is static and fully known to the planner. To address the first point, we could consider the notion of commit times proposed by Karaman *et al.* in [3]. The idea is that the motion planner will commit to the execution of a sub-optimal path segment for some amount of time and, during that execution, it will continue to refine the trajectory using the terminal configuration of that path segment as a new starting configuration. By improving the path during navigation, the overall time required could potentially be reduced. In this case, actions could be reformulated as real valued commit times and we could use policy gradient methods [4, chapter 13] designed for continuous action spaces. However, we might consider a situation in which the planner commits to execution along a highly suboptimal path. Further planning might reveal that the best action would be to turn around and go back the way it came. In such a situation, rather than committing to the initial trajectory, some amount of stationary computation would have been the best initial choice. By this reasoning, we would want to include both commit times and stationary computation times in our action space. This is a much more dynamic problem which falls outside the scope of this work, but it may well be that with increased complexity in terms of the action space, the margin for improvement over a fixed policy can be increased.

To address the second point, we could consider cases in which the environment is not fully known because of non-stationary obstacles or an incomplete map. Such environments are more interesting as they reflect many real world applications. One such algorithm for tackling these sorts of environments is ADA* [11]. As an extension of the ARA* algorithm used in this work, the same concepts of inconsistency are applied and so a similar, if not identical, formulation of planner state as given in Chapter 3 could be applied. We believe that the learning framework developed in this work could be readily applied to uncertain environments using ADA* as a trajectory planner.

As a final consideration, we are interested in how our learning framework would generalize to other domains with high dimensional state spaces such as robot arms with many degrees of freedom. The interplay between computation time and the time required to carry out a solution is inherently domain specific, so the utility of our approach in such situations is not immediately clear. To explore this we would need only a suitable set of motion primitives and an appropriate cost function. This is left for future investigation.

6. SUMMARY

In this work, we have presented a motion planner which learns to efficiently manage its time while planning and traversing a variety of simulated environments. This is accomplished using anytime motion planning techniques on state-lattices and reinforcement learning techniques. With our approach, we achieve a performance gain over best case fixed time planning policies and come close to matching the behavior of an ideal decision making agent. We have also shown how a generally trained planner can perform well across multiple environments without modification.

REFERENCES

[1] S. M. LaValle, *Planning Algorithms*. Cambridge University press, 2006.

[2] M. Likhachev, G. J. Gordon, and S. Thrun, “ARA*: Anytime A* with Provable Bounds on Sub-optimality,” in *Advances in Neural Information Processing Systems*, pp. 767–774, MIT Press, 2003.

[3] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, “Anytime Motion Planning Using the RRT,” in *2011 IEEE International Conference on Robotics and Automation*, pp. 1478–1483, IEEE, 2011.

[4] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. MIT Press, 1998.

[5] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*. Springer, 2016.

[6] L. Kavraki, P. Svestka, and M. H. Overmars, “Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces,” *International Transactions on Robotics and Automation*, vol. 12, pp. 566–580, 1996.

[7] J. J. Kuffner and S. M. LaValle, “RRT-Connect: An Efficient Approach to Single-Query Path Planning,” in *2000 IEEE International Conference on Robotics and Automation*, pp. 995–1001, IEEE, 2000.

[8] M. Pivtoraiko and A. Kelly, “Generating near minimal spanning control sets for constrained motion planning in discrete state spaces,” in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, pp. 3231–3237, 2005.

[9] “Search-Base Planning Lab.” `http://www.sbpl.net/`.

[10] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, 1968.

[11] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An Anytime, Replanning Algorithm," in *Proceedings of the International Conference on Autonated Planning and Scheduling*, pp. 262–271, 2005.

[12] M. Likhachev and D. Ferguson, "Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles," *International Journal of Robotics Research*, vol. 28, pp. 933–945, 2009.

[13] G. Konidaris, S. Osentoski, and P. S. Thomas, "Value Function Approximation in Reinforcement Learning Using the Fourier Basis," *AAAI Conference on Artificial Intelligence*, vol. 6, 2011.

[14] S. Papierok, A. Noglik, and J. Pauli, "Application of Reinforcement Learning in a Real Environment Using an RBF Network," in *Proceedings of the International Workshop on Evolutionary Learning for Autonomous Robot Systems*, 2008.

[15] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the Game of Go Without Human Knowledge," *Nature*, vol. 550, pp. 484–503, 2016.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *NIPS Workshop on Deep Learning*, 2013.

[17] D. E. Knuth, *The Art of Computer Programming; Volume 2: Seminumeral Algorithms*. Addison-Wesley, 1981.