2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

Learning to Use Adaptive Motion Primitives in Search-Based Planning for Navigation

Raghav Sood, Shivam Vats and Maxim Likhachev

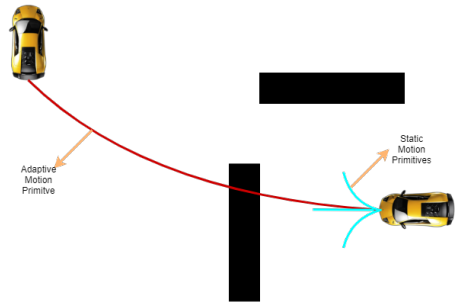
***Abstract*— Heuristic-based graph search algorithms like A* are frequently used to solve motion planning problems in many domains. For most practical applications, it is infeasible and unnecessary to pre-compute the graph representing the whole search space. Instead, these algorithms generate the graph incrementally by applying a fixed set of actions (frequently called *motion primitives*) to find the successors of every node that they need to evaluate. In many domains, it is possible to define actions (called *adaptive* motion primitives) that are not pre-computed but generated on the fly. The generation and validation of these adaptive motion primitives is usually quite expensive compared to pre-computed motion primitives. However, they have been shown to drastically speed up search if used judiciously. In prior work, ad hoc techniques like fixed thresholds have been used to limit unsuccessful evaluations of these actions. In this paper, we propose a learning-based approach to make more intelligent decisions about when to evaluate them. We do a thorough empirical evaluation of our model on a 3 degree-of-freedom (dof) motion planning problem for navigation using the Reeds-Shepp path as an adaptive motion primitive. Our experiments show that using our approach in conjunction with search algorithms leads to over 2x speedup in planning time.**

I. INTRODUCTION

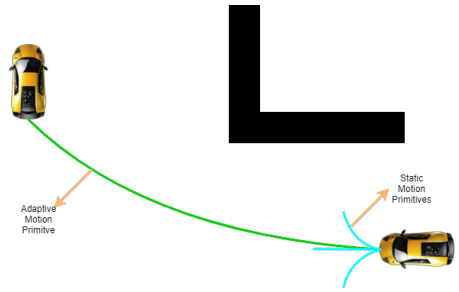
Graph search based methods are commonly used for solving a host of robot motion planning problems. These methods represent the search space as a graph, wherein each vertex is a valid state of the robot in the configuration space (C-space) and an edge between two vertices corresponds to a valid motion between those states. The graph is usually generated by applying a small set of pre-computed actions, called *motion primitives* at every state. In addition, we can also use *adaptive* motion primitives [1] which are actions that are generated on the fly. In many domains, they have proven to be useful in making motion planning more efficient and precise. Adaptive motion primitives are usually computed by running a solver either analytical or numerical and as a result, can be much more computationally expensive than a pre-computed primitive. Hence, using these adaptive motion primitives naively may sometimes even slow down the search substantially.

One standard approach for minimizing the calls to compute adaptive motion primitives is using them with some fixed preconditions. For example, in navigation, one may try to add an edge to the goal corresponding to the Dubins [2] [3] or Reeds-Shepp [4] path, if a state is close enough to the goal. The problem with this approach is that the decision regarding choosing the threshold distance is not well informed and doesn't take into account information about robot surroundings with context to the robot pose. As we can see in Fig. 1 (b), an adaptive motion primitive can make search progress much faster if used judiciously. However, in Fig. 1 we can observe that even though in both the cases the distance between two poses being connected by the primitive is same, the first one is invalid whereas the second one is valid. This leads to the insight that the decision boundary of the validity of motion primitives is not trivial and depends upon the pose of the robot and obstacles in its surroundings. Fig. 2 shows the decision boundary for the validity of a Reeds-Shepp motion primitive to a fixed goal pose from all other possible poses of the robot in 2 dimensions. This shows that activating this primitive only within a fixed radius from the goal is not a very informed approach and we can make better decisions regarding adaptive motion primitive calls if we had a way to learn this decision boundary.

This work approaches the bottleneck of expensive generation and validation of adaptive motion primitives with two



(a) Infeasible adaptive motion primitives slow down search due to long validation time



(b) Feasible adaptive motion primitives accelerate search

Fig. 1: Trade-off associated with using adaptive motion primitives

[1]Raghav Sood, Shivam Vats and Maxim Likhachev are with the Robotics Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA {raghavso,svats,mlikhach}@andrew.cmu.edu

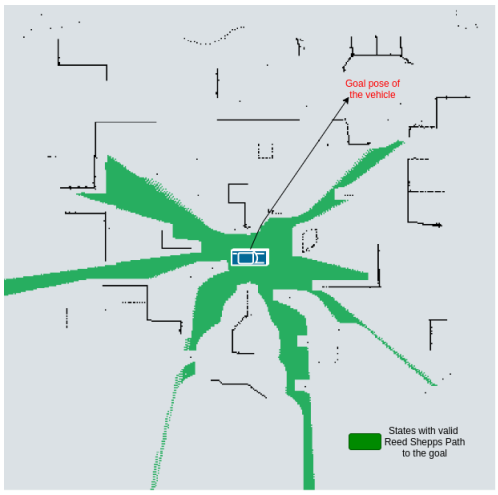


Fig. 2: Non-trivial decision boundary for validity of Reeds-Shepp path for a given goal. The boundary is drawn for a fixed heading of 0 degrees for the start pose of the vehicle. The green area shows the states from which we can connect to goal with a valid Reeds-Shepp motion primitive.

key insights. First, the inherent structure of the environment and C-space of the robot can be used to make informed decisions on calls to adaptive motion primitives. Second, using a parallel neural network inference for the task can substantially speed up the pruning of invalid adaptive motion primitives.

The first insight leads to the idea of learning to predict the validity of adaptive motion primitives using some representation of the immediate environment of the robot for a given robot state. Provided enough examples, the robot can learn to predict the validity of these primitives without explicitly generating and validating them. Multilayer feedforward neural networks are universal approximators and can learn arbitrarily accurate representation of a continuous function[5]. In this work, we train a deep neural network, AMPNet to predict the validity of adaptive motion primitives using computationally expensive motion primitive generator and evaluator for training.

The second insight leads to the idea of leveraging the efficiency of neural networks at processing large batches of data. Using a neural network for inferring the validity of an adaptive motion primitive is still an expensive operation due to additional computational overheads. Therefore, we avoid inferring the validity of an adaptive motion primitive for each state expansion during the search by doing a pre-processing step before search. We intelligently sample our C-space and do a batch inference for determining the validity of adaptive motion primitives from sampled states to a fixed successor state and later use this information during search for determining the validity of the adaptive motion primitive from the current state being expanded in search.

We test our approach on 3-DOF navigation domain for a non-holonomic ground robot with Reeds-Shepp path as an adaptive motion primitive. We compare our method to the one used in [1] that uses adaptive motion primitives to snap to the goal only when the state to be expanded is within some fixed distance from the goal.

II. RELATED WORK

Several previous works have attempted to speed up motion planning using learning-based techniques. Most of these works focus on speeding up sampling based motion planning methods like RRT [6] and PRM [7]. Collision checking is the primary bottleneck in sampling based motion planning. Therefore, there has been a lot of work in speeding up collision checking using learned models for collision detection [8] [9] and reducing collision checks required by biased sampling using learned models [10]. Our work differs from this class of work in two ways. First, our focus is on speeding up heuristic search-based motion planning. Heuristic search based motion planning allows the incorporation of complex cost functions and constraints, and provides consistent plans with theoretical guarantees such as completeness and sub-optimality bounds [11]. Second, we are not trying to learn a proxy collision checking model. Instead, we are trying to predict the validity of an adaptive motion primitive without even generating it.

There has also been some work in speeding up search based motion planning on graphs with expensive edges by using lazy evaluations. Algorithms like Lazy Weighted A* [12] and LazySP [13] try to reduce the number of edge evaluations by postponing the evaluation of an edge until it is absolutely necessary to evaluate it during the search. This comes with the overhead of additional graph operations which can be large depending on the look-ahead. Our work is complementary to these algorithms. We don't consider evaluating an edge that corresponds to an adaptive motion primitive if our predictor predicts it to be invalid.

III. ALGORITHM

We present AMPNet, a feed-forward neural network that predicts validity of an adaptive motion primitive, and its use within a search based planning algorithm. AMPNet is a multilayer perceptron that takes as input, the local obstacle information around the robot combined with the relative position of the goal with respect to robot state. It predicts if an adaptive motion primitive from the robot state to the goal is feasible or not.

We also present an algorithm that takes advantage of the fact that inference in neural networks is highly parallelizable using batches. This algorithm combined with a weighted A* [14] search then explores the configuration space of the robot and returns a valid path with bounds on suboptimality.

A. AMPNet

1) Data Generation: The input features to the network are divided into two parts. The first part of the input encodes the local information about the obstacles in the vicinity of the current robot state $S_{curr} = \{X_{curr}, Y_{curr}, \theta_{curr}\}$ and the second part encodes the relative pose of the goal of the motion planning problem S_{goal} with respect to S_{curr}. For the

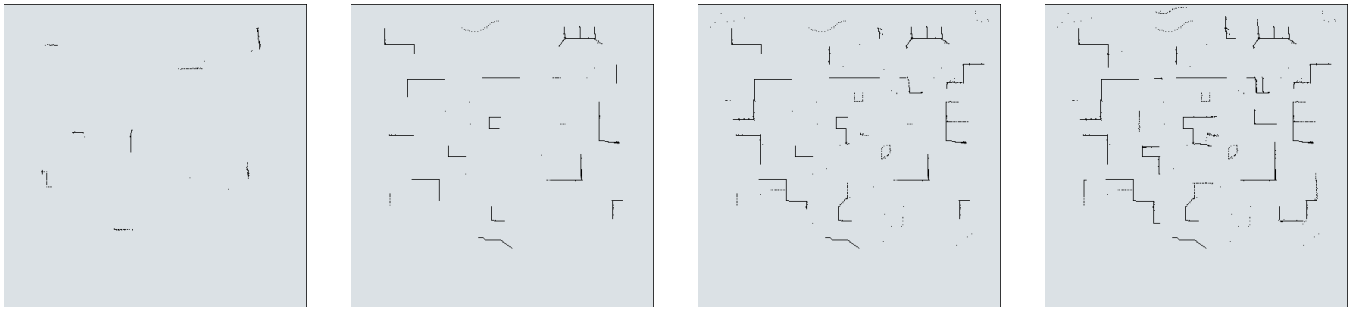


Fig. 3: Environments with increasing complexity(from left to right) with respect to number and shape of obstacles

first part, we choose to do a 360 degrees ray tracing from S_{curr} and get a vector of distances $\boldsymbol{d}$ to the closest obstacle in each direction. For the second part we calculate relative pose $\boldsymbol{p_{rel}}$ of S_{goal} with respect to S_{curr} in polar coordinates with the reference angle aligned with θ_{curr}. To estimate the validity $\boldsymbol{v}$ of a primitive from S_{curr} to S_{goal}, we use Reeds-Shepp algorithm to first generate the adaptive motion primitive and then validate it using a collision checker. The data collection process then gathers triplets of ray tracing distance vectors $\boldsymbol{d}$, relative start goal poses $\boldsymbol{p_{rel}}$ and validity of primitive $\boldsymbol{v}$ into a dataset $\mathcal{D} = \{\boldsymbol{d}, \boldsymbol{p_{rel}}, \boldsymbol{v}\}$. The reason for choosing a 360 degree ray tracing distance vector as an input feature is motivated by the fact that this feature is generalizable and can work on any arbitrary environment. At the same time, the relative position of S_{goal} with respect to S_{curr} in polar coordinates helps us to compactly represent the relative pose of states being connected by the motion primitive.

2) AMPNet Training: We train AMPNet using the generated dataset $\mathcal{D}$. The network takes the ray tracing distance vector $\boldsymbol{d}$ and relative pose $\boldsymbol{p_{rel}}$ of S_{goal} with respect to S_{curr} as input and outputs if the adaptive motion primitive from S_{curr} to S_{goal} is valid or not. As the motive of using the network is to speed up validation of adaptive motion primitives by pruning away potentially invalid primitives, we impose a limit on the inference time for the network. A network with inference time considerably lower than the actual generation and validation time of the primitive is required for the task. This, in turn, puts a limit on the complexity of the network. The network is required to be as simple as possible while achieving a good enough accuracy to guarantee speedups in planning times. Furthermore, false-negative inferences affect the planning times much more than false-positives because a large number of false negatives can lead us to discard some valid adaptive motion primitives. Therefore, during training, we also keep in mind that we want to penalize false negative inferences more as compared to false positive inferences. The network is designed keeping these trade-offs in mind. The network consists of one input layer, 2 fully connected hidden layers followed by dropout and an output layer. The network training minimizes the weighted binary cross entropy loss.

B. Pre-Processing before Search

Using a neural network inference for validating an adaptive motion primitive during each expansion of search can be expensive. This is due to the computational overhead associated with the generation of inputs and forward pass through the network. The fact that batch inference in neural networks is highly parallelizable combined with the observation that inference for neighboring robot poses in C-space is quite likely to be similar helps us to come up with a more efficient algorithm that can considerably speed up the search.

Algorithm 1 Pre-Processing before Search

Input: S_{goal}: End vertex of the Adaptive Motion Primitive in C-space
Input: N_{sample}: Number of vertices to be sampled in C-space C
Input: AMP_{net}: Learned model for evaluation of adaptive motion primitives
Output: M_{inf} : Map from sampled vertices to adaptive motion primitive validity as inferred from the model
Output: K_t: k-d tree comprising of all sampled vertices

1: $P_{sample} \leftarrow$ sample N_{sample} vertices from C.
2: Calculate Features F for all samples in P_{sample}
3: $F_{batch} \leftarrow$ all features F stacked in a batch.
4: $I_{batch} \leftarrow$ Inference for forward pass of F_{batch} through AMP_{net}
5: $M_{inf} \leftarrow$ Create a Map from P_{sample} to I_{batch}
6: $K_t \leftarrow$ create a k-d tree containing P_{sample}
7: **return** M_{inf}, K_t

We propose a pre-processing step before the search as described in algorithm 1. First, we sample N_{sample} vertices in the C-space of the robot and store them in P_{sample}. We then calculate the features F for all vertices in P_{sample} and store them in F_{batch}. Next, we pass F_{batch} through AMPNet for inferring validity of all the edges from P_{sample} to S_{goal}. The output of the network I_{batch} is stored in a hash map M_{inf} from P_{sample} to I_{batch}. We then create a k-d tree K_t for storing all vertices in P_{sample}. The map M_{inf} and the k-d tree K_t are used during the search for pruning invalid adaptive motion primitives.

Sampling Strategy: We come up with a sampling strategy

to maximize the efficiency of our algorithm given a fixed number of samples. Given N samples we do sampling in 2 rounds. The algorithm first samples a proportion of vertices uniformly in the C-space of the robot and does a batch inference for the validity of motion primitives for these sampled vertices. In the next round, we go through all the vertices with valid inferences from the previous round and sample new samples uniformly within a fixed radius around them.

C. Adaptive Motion Primitive Validation during Search

Using the results of the pre-processing step, we can use adaptive motion primitives during all expansions in the search using the procedure outlined in algorithm 2. For given state S_{curr} being expanded in search, we query K nearest neighbours $\{S_{n1}, S_{n2}.., S_{nk}\} \in P_{sample}$ using K_t. We check for the validity of adaptive motion primitives from all of these neighbors by querying M_{inf}. Then we do a poll amongst the neighbors and select the most common inference for determining the validity of adaptive motion primitive from S_{curr}.

Algorithm 2 Adaptive Motion Primitive Validation during Search using AMPNet

Input: K_t: k-d tree comprising of all sampled vertices.
Input: S_{curr}: Current state being expanded in search.
Input: M_{inf}: Map from sampled vertices to adaptive motion primitive validity as inferred from the model.
Output: I: Boolean variable indicating predicted validity of the edge.

1: $\{S_{n1}, S_{n2}.., S_{nk}\} \leftarrow$ query K nearest neighbour for S_{curr} from K_t
2: $I_{neigh} \leftarrow$ lookup inferences for $\{S_{n1}, S_{n2}.., S_{nk}\}$ from M_{inf}
3: $I_{curr} \leftarrow$ Mode inference in I_{neigh}
4: **return** I_{curr}

D. Search

$GetSuccessors$ is a standard routine used in search based motion planners. The routine determines the successor states $Succs$ that can be reached from S_{curr} given the set of actions or motion primitives. Algorithm 3 outlines the modified $GetSuccessors$ procedure we use to incorporate our algorithm within a heuristic search. $GetSuccessors$ takes S_{curr} and set of motion primitives as input. Here we have 2 sets of motion primitives, M_{static} and $M_{adaptive}$. The modification that we make here is that if a motion primitive is adaptive, we call our validation network AMPNet to validate it as outlined in algorithm 2. If the network predicts the motion primitive to be invalid we do not add it to our list of successors $Succs$. However, if the network predicts the motion primitive to be valid, we generate it and validate it using a collision checker before adding it to the list of successors. This step is required as the learned model is approximate and we need to discard all invalid motion primitives.

Algorithm 3 Modified Get Successors Routine for Search

1: **procedure** GETSUCCS(S_{curr}, M_{static}, M_{adap})
2: $Succs \leftarrow \{\}$
3: **for** all actions $a_{static} \in M_{static}$ **do**
4: $S_{succ} \leftarrow$ *apply* (S_{curr},a_{static})
5: **if** *collision check* (S_{curr},a_{static}) **then**
6: $Succs \leftarrow Succs \cup \{S_{succ}\}$
7: **end if**
8: **end for**
9: **for** all actions $a_{adap} \in M_{adap}$ **do**
10: **if** *valid_AMPNet* (S_{curr},a_{adap}) **then**
11: $S_{succ} \leftarrow$ generate adaptive motion primitive for S_{curr}
12: **if** *collision check* (S_{curr},a_{static}) **then**
13: $Succs \leftarrow Succs \cup \{S_{succ}\}$
14: **end if**
15: **end if**
16: **end for**
17: **return** $Succs$
18: **end procedure**

IV. RESULTS

Experimental Setup: We evaluate AMPNet and our algorithm combined with a weighted A* search on a 3-DoF motion planning problem for a non-holonomic ground robot. We use a set of 4 static motion primitives combined with Reeds-Shepp path as an adaptive motion primitive. Reeds-Shepp path is defined as the shortest traveling path of the Reeds-Shepp Car, a car that can go both forward and backward with a constrained turning radius. Reeds-Sheep path can be viewed as a generalized version of Dubins path [3]. We evaluate the performance of our algorithm on environments with varying order of complexity in regards to the number and positions of obstacles as shown in figure 3. The environment is parameterized by location of obstacles and the origin. The input to AMPNet is of length 47. The first 45 inputs correspond to 360 degree ray tracing around the vehicle's current state. The discretization for the ray tracing angle is 8 degrees i.e each ray is 8 degrees apart. The last 2 inputs are the relative pose of the goal with respect to the current state of the robot in polar coordinates. The number of samples used in the pre-processing step of our algorithm is chosen to be 5000 and the value of K for nearest neighbor lookup is set to be 3.

A. AMPNet Results

Accuracy : Table II summarizes the accuracy of AMPNet. As the network was trained with a weighted binary cross entropy loss to penalize misclassification of positive class more, we can see that the network achieves an accuracy of 97% on the positive class and is unlikely to misclassify valid motion primitives. False Positives are handled by our validation step in the algorithm and don't lead to invalid paths. However, a higher number of false positives means more validation checks.

TABLE I: Summary of path characteristics and planning statistics for AMPNet algorithm and baseline methods averaged over 100 planning queries across 7 environments with varying degree of complexity

Method	Planning Time(s)	Std. Deviation	Max Time(s)	Path Cost	Expansions per sec
Reeds-Shepp with AMPNet	0.37	1.10	6.1	338	57578
Reeds-Shepp within a fixed distance	0.91	14.5	22.1	342	105790
Reeds-Shepp always	0.98	15.2	23.27	336	38151
No Reeds-Shepp	1.67	25	29	345	157600

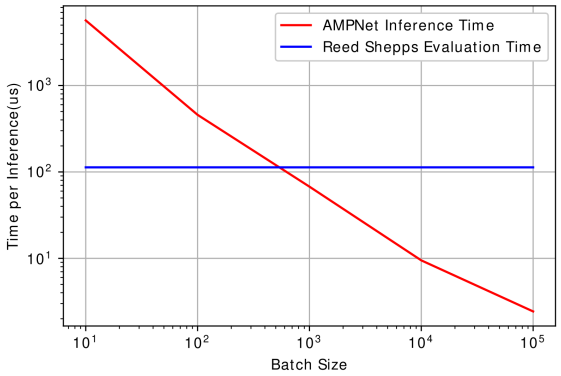


Fig. 4: Time per inference vs batch size for AMPNet

TABLE II: AMPNet Accuracy Stats.

Accuracy	False Positives	False Negatives
87%	20%	3%

Inference Time: Figure 4 shows time per inference vs batch size for AMPNet. Note that both the axes of the plot are in logarithmic scale. We can see that time per inference falls exponentially with the increase in batch size. We use this property to our advantage during the pre-processing step mentioned in algorithm 1. We observe from the figure that if we use a batch size greater than 800 we can do much better than the actual evaluation time of Reeds-Shepp path using the classical approach.

B. Planning Results

We compare our method labeled as Reeds-Shepp with AMPNet in Table I with 3 baselines: Weighted A* planner with no adaptive motion primitive labeled as No Reeds-Shepp, weighted A* planner using Reeds-Shepp path as adaptive motion primitive used always at each expansion during search labeled as Reeds-Shepp always and weighted A* planner that uses Reeds-Shepp motion primitive only when the state to be expanded during search is within a fixed distance from the goal labeled as Reeds-Shepp within a fixed distance. AMPNet was implemented in C++ using LibTorch api from PyTorch [15]. All computations were done using CPU only on a 16GB memory, intel i7(gen 9) machine.

We evaluate all methods on 7 environments with varying degrees of complexity with 100 start and goal pair configurations for each environment. The timeout for solving the problem is set to be 30 seconds. The results for the tests have been summarized in Table I. Reeds-Shepp with AMPNet is able to solve queries **2.5x** faster than Reeds-Shepp at fixed distance, **2.7x** faster than naive Reeds-Shepp used always, and **4.5x** faster than not using Reeds-Shepp at all.

Variation with Environment Complexity: As mentioned, we test all methods on environments with varying degrees of complexity. Environment specific results for 3 environments: uncluttered, slightly cluttered and highly cluttered are shown in figure 5. There are high chances of Reed-Shepps motion primitive to be valid from long distances in the uncluttered environment. This can be observed from figure 5(a) where our algorithm and the method where we use Reeds-Shepp always, perform an order of magnitude faster than other methods. In slightly cluttered environments, we can observe that the method involving using Reeds-Shepp always doesn't perform well as the chances of the primitive to be valid from an arbitrary far distance are very low. Our algorithm still performs better than other methods because it can intelligently use the adaptive motion primitives and only tries to extend a primitive when there are high chances of it being valid. This fact is further corroborated by the performance of our algorithm on highly cluttered environments as shown in figure 5 (c). We can observe that the performance gap between the method using Reeds-Shepp at distance and our algorithm is not as pronounced as in other environments because the chances of a Reeds-Shepp primitive to be valid are very low in highly cluttered environments.

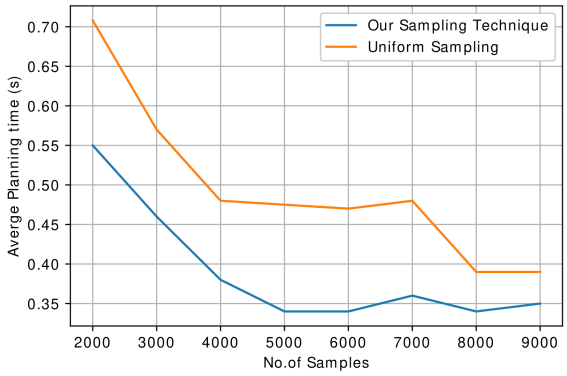
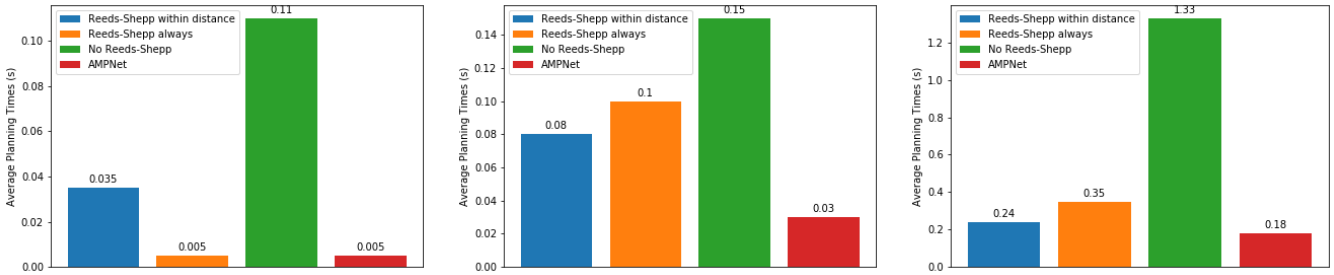


Fig. 6: No. of samples vs average planning times

Effect of Number of Samples: As outlined in algorithm 1, the number of samples N_{sample} is an important parameter. We tested our algorithm's performance on a varying number



(a) Planning time comparison in uncluttered environment

(b) Planning time comparison in slightly cluttered environment

(c) Planning time comparison in highly environment

Fig. 5: Performance of tested methods with varying complexity of environments

of samples using 2 sampling strategies. In the first strategy, we naively sample the C-space uniformly and use those samples for the pre-processing step in the algorithm. The second strategy has been outlined in section III-B. We vary the number of samples from 2000 to 9000 and the results of the performance of both the strategies are shown in figure 6. It can be observed that our sampling strategy achieves lower average planning times with much fewer samples compared to the naive approach.

C. Additional Analysis of Run-Time

The generation and validation time T_{val} for Reeds-Shepp path is 0.113ms on average. In contrast, the validation time using AMPNet T_{net} is 0.0035 ms. Considering that we only evaluate primitives that are labeled positive by the network, the validation time for a positive inference is

$$T_{pos} = T_{net} + T_{val} \tag{1}$$

Whereas, validation time for a negative inference is

$$T_{neg} = T_{net} \tag{2}$$

Considering AMPNet's accuracy on positive class P_{pos} as shown in table II, the average expected evaluation time for a true valid adaptive motion primitive is

$$T_{exp}^{pos} = P_{pos}(T_{pos}) + (1 - P_{pos})(T_{neg}) \tag{3}$$

Similarly, average expected time for true invalid adaptive motion primitives with negative class accuracy P_{neg} is

$$T_{exp}^{neg} = P_{neg}(T_{neg}) + (1 - P_{neg})(T_{pos}) \tag{4}$$

Empirical probability P_{val} of a valid Reeds-Shepps path in our dataset $\mathcal{D}$ is found out to be 0.25. Therefore average expected evaluation time T_{exp}^{eval} for a Reeds-Shepp path using AMPNet can be calculated as

$$T_{exp}^{eval} = P_{val}T_{exp}^{pos} + (1 - P_{val})T_{exp}^{neg} \tag{5}$$

The results of the above equation have been summarized in table III

TABLE III: Run-Time Analysis Results

T_{pos}	0.1165 ms
T_{neg}	0.0035 ms
T_{exp}^{pos}	0.1131 ms
T_{exp}^{neg}	0.0261 ms
T_{exp}^{eval}	0.0282 ms

We can see from table III that average expected time per evaluation using AMPNet and our algorithm is **4.2x** lower than the actual evaluation time for the dynamic motion primitive.

V. CONCLUSION

In this paper we present a technique for intelligently using adaptive motion primitives in heuristic search based algorithms for motion planning. Our method uses a learned model to approximate the decision boundary of the validity of adaptive motion primitives. The model uses information about the robot's surroundings and the robot's pose to learn the decision boundary. This approach of using adaptive motion primitives considerably speeds up motion planning. The approach can be extended to other motion planning domains where adaptive motion primitives are used and are expensive to evaluate during the search. Furthermore, this approach can also be extended to sampling based planners like RRT [6] or PRM [7] where we can use the learned model to find feasibilty of an edge extension from tree to a new sampled vertex. Future work includes applying the idea of learning a predictor for adaptive motion primitives to other domains including planning for manipulation and kinodynamic planning.

VI. ACKNOWLEDGMENT

This work was supported by ONR grant N00014-18-1-2775.

REFERENCES

[1] B. Cohen, G. Subramanian, S. Chitta, and M. Likhachev, "Planning for manipulation with adaptive motion primitives," pp. 5478 – 5485, 06 2011.

[2] R. Pěnička, J. Faigl, P. Váňa, and M. Saska, "Dubins orienteering problem," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 1210–1217, 2017.
[3] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents," *American Journal of Mathematics*, vol. 79, no. 3, p. 497, jul 1957. [Online]. Available: https://doi.org/10.230
[4] J. A. Reeds and L. A. Shepp, "Optimal paths for a car that goes both forwards and backwards." *Pacific Journal of Mathematics*, vol. 145, no. 2, pp. 367–393, 1990. [Online]. Available: https://projecteuclid.org:443/euclid.pjm/1102645450
[5] K. Hornik, M. Stinchcombe, H. White *et al.*, "Multilayer feedforward networks are universal approximators." *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
[6] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.
[7] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, pp. 566 – 580, 09 1996.
[8] N. Das and M. Yip, "Learning-based proxy collision detection for robot motion planning applications," 2019.
[9] J. C. Kew, B. Ichter, M. Bandari, T.-W. E. Lee, and A. Faust, "Neural collision clearance estimator for fast robot motion planning," 2019.
[10] Jinwook Huh and D. D. Lee, "Learning high-dimensional mixture models for fast collision detection in rapidly-exploring random trees," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 63–69.
[11] M. Likhachev, G. Gordon, and S. Thrun, "Ara*: Anytime a* with provable bounds on sub-optimality," in *Proceedings of the 16th International Conference on Neural Information Processing Systems*, ser. NIPS'03. Cambridge, MA, USA: MIT Press, 2003, p. 767–774.
[12] B. Cohen, M. Phillips, and M. Likhachev, "Planning single-arm manipulations with n-arm robots," 07 2014.
[13] C. Dellin and S. Srinivasa, "A unifying formalism for shortest path problems with expensive edge evaluations via lazy best-first search over paths with edge selectors," 03 2016.
[14] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artif. Intell.*, vol. 1, pp. 193–204, 12 1970.
[15] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.