Faster Motion Planning Using Learned Local Viability Models

Maciej Kalisiak
Dept. of Computer Science, University of Toronto
mac@dgp.toronto.edu

Michiel van de Panne
Dept. of Computer Science, University of British Columbia
van@cs.ubc.ca

***Abstract*— Current motion planners, in general, can neither “see” the world around them, nor learn from experience. That is, their reliance on collision tests as the only means of sensing the environment yields a tactile, myopic perception of the world. Such short-sightedness greatly limits any potential for detection, learning, or reasoning about frequently encountered situations. As a result, it is common for current planners to solve and re-solve the same general scenarios over and over, each time none the wiser. We thus propose a general approach for motion planning, as well as a specific illustrative algorithm, in which local sensory information, in conjunction with prior accumulated experience, are exploited to improve planner performance. Our approach relies on learning viability models for the agent’s “perceptual space”, and the use thereof to direct planning effort. Experiments with three test agents show significant speedups and skill-transfer between environments.**

I. Introduction

In general, current motion planners rely solely on collision detection for sensing the surrounding environment, which leads to very tactile and myopic perception. As a result, such planners cannot detect, learn, nor reason about commonly occurring patterns or scenarios, and instead end up solving such problems “from scratch” each time. For example, there is usually no substantial difference between the first and the hundredth time that a typical planner solves a general parallel parking problem. In contrast, the ideal motion planner would notice general patterns in the problem space, and through observing its own solutions, it would learn corresponding general motion strategies. These could be employed in subsequent queries, leading to motion planning in higher and more macroscopic terms.

This paper is only the first step towards that ideal. We propose a weaker version of this lofty goal, in which the planner in effect learns what nonviable scenarios look like, and then avoids them in subsequent planning. Three components make up our approach. First, we give the planner the ability to “see” by instrumenting the agents with rangefinder-like virtual sensors (see Figure 1). This allows one to describe the agent’s state in more local, “perceptual” terms. Second, a large body of successful, perceptually-parameterized motions is accumulated, either from the planner’s own previous solutions, or from an external source. Finally, this knowledge is then exploited in further queries, whereby the planner’s search is limited to motions resembling previous exemplars of successful motions. Sections III through V describe these three parts further.

The benefit of this approach, which we call planning with “*viability filtering*”, is reduced planning time, resulting from

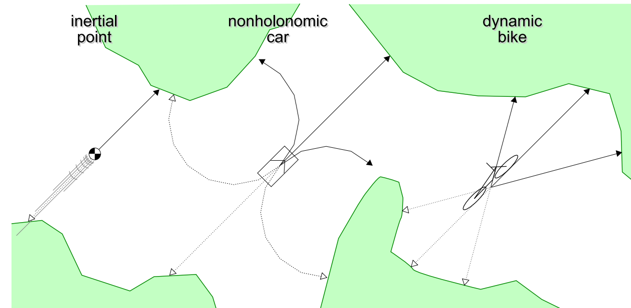

Fig. 1. Agents used in experiments. Lines indicate (virtual) agent-mounted range sensors used in planning. They are discussed in §VII.

forestalling planner exploration in nonviable regions of the search space. That is, by in effect preemptively culling the planner’s search space of regions which cannot possibly lead to a solution (or are very unlikely to do so), we ensure that all planner effort is spent more productively. Nonviable regions represent wasted effort since a nonviable agent state, if not already in collision, must then unavoidably lead to failure, yet no valid solution will ever entail a collision prior to reaching the goal.

The approach presented in this paper should integrate well with many existing planners. In general, our approach is applicable to any iterative planner for actuated agents in which exploration proceeds by repeated choosing of a control action to apply next, from among some finite set of control actions, and thus where such preemptive pruning makes sense. This paper, for example, demonstrates the use of this method with the RRT-Blossom [1] planner; section VI gives the implementation details, while section VII outlines achieved results.

II. Previous work

Motion planning has many real world applications, and is thus an active area of research. Comprehensive overviews are provided by Latombe [2], and more recently by LaValle [3], while Dean and Wellman [4] provide a more unified view of motion planning and control.

A milestone in the history of motion planning is the introduction of (stochastic) sampling-based planners. An early example is the Randomized Path Planner (RPP) [5], which performs gradient descent down a potential field that spans the environment’s free-space, and backtracks when local minima are encountered. A more robust and current alternative is proposed in [6], [7], in the form of Rapidly-exploring Random Trees (RRTs). These planners work by

quickly exploring the free-space with tree structures; commonly two trees are used, one rooted at the starting state, the other at the goal, and a solution is found when the trees meet. Probabilistic Roadmaps (PRMs) [8], on the other hand, build graphs of reachable configurations. This is achieved by stochastically picking random "milestones" and connecting them together using a local planner, or in the case of simple agents, using straight lines. Queries are solved by connecting the start and goal to the roadmap and finding the shortest path through the resultant graph.

The bulk of these planners concern themselves with the agent's kinematics. *Kinodynamic* motion planning, on the other hand, takes also into account the agent's dynamics. In practical terms, the former planners work with the agent's configuration q, whereas kinodynamic ones use the agent's state x, where usually $x = (q, q')$. Aside from the larger search space, the planner's task is often made more difficult by an abundance of deep "dead-ends" in this space[1], the result of the "cross product" of the agent's laws of motion and the imposed constraints (e.g., environment geometry). The approach introduced in this paper is particularly well suited to these types of problems. The first such kinodynamic planner, presented in [10], uses a deterministic, brute force search over a grid-discretized state space, and hence is feasible only for trivial systems. RRT is extended to kinodynamic planning in [9], and proves to be a more scalable solution. Further work addresses some of its remaining limitations: RRT with Collision Tendencies (RRT-CT) in [11] aims to mitigate RRT's sensitivity to the distance metric used in growing the trees, while RRT-Blossom [1] addresses the planner's poor performance in highly constrained environments.

We are unaware of motion planners that, in an environment-independent fashion, learn from observing their own operation. Some planners do accumulate experience, [12] for example, but it is then applied to queries in the *same* environment.

Viability is a key underlying concept in our work. Briefly, a *viable* or *controllable* state is one from which the subject can, through application of appropriate control actions, remain indefinitely free of collision or failure. Conversely, a *nonviable* state is one in which the subject has passed the "point of no return", where failure is no longer avoidable, even though it may be postponable for some finite time. Figure 2 illustrates these concepts, while a more thorough treatment can be found in [13], [14]. Our work also deals with reachability, a complementary concept to viability; a comprehensive overview of the topic can be found in [15]. There has been recent interest in applying viability to control, such as [13], [16], but we are not aware of any such work in the general area of motion planning.

[1]Previous work, such as [9], refers to these as *regions of inevitable collision*, $\mathcal{X}_{ric}$. These dead-ends hinder tree-based planners in much the same way that local minima hinder potential field planners.

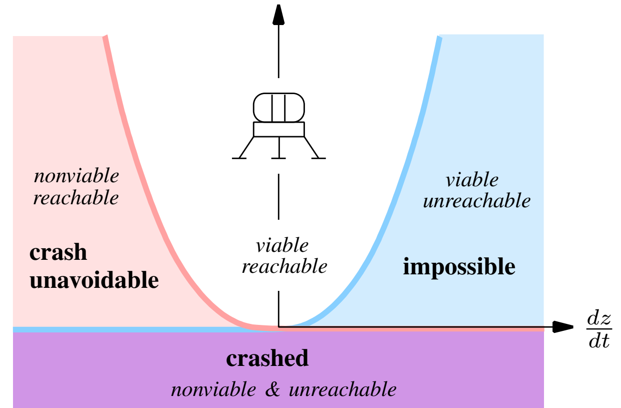

Fig. 2. An example of viability and reachability, using lunar lander; z is altitude. The "crash unavoidable" region is nonviable since the downward velocity exceeds the braking power of the lander's bounded thrust. The "impossible" region is unreachable since the upward velocity exceeds what can be achieved, even if maximal thrust is applied from altitude $z = 0$.

III. Sensing & situation encoding

Traditionally, motion planners treat x, the state of the agent, and e, that of the environment[2], as separate entities, and only bring them together during collision checks. In the following it is useful to define the combined, full *system state*, $x^+ = (x, e)$. The corresponding state spaces are analogously combined: if $x \in \mathcal{X}$ and $e \in \mathcal{E}$ then $x^+ \in \mathcal{X}^+$, $\mathcal{X}^+ = \mathcal{X} \times \mathcal{E}$.

Global descriptors such as x^+ provide poor generalization potential since any acquired knowledge so parameterized does not transfer well to other, even very similar environments. A more local projection of the agent's state will generally be more suitable. The latter can be attained by equipping the agent with virtual *sensors* (e.g., Figure 1), and combining the resultant synthetic local information with a subset of x^+.

More formally, a sensor σ is a function which maps the subject's state and the environment geometry to a scalar value:

$$\sigma(x^+) : \mathcal{X}^+ \rightarrow \mathbb{R}$$

For a particular system state $x^+ \in \mathcal{X}^+$, one can compute all the sensor values and concatenate them into a single vector,

$$s = \big(\sigma_1(x^+),\ \sigma_2(x^+),\ \ldots\big), \quad s \in \mathcal{S}$$

which we refer to as the *sensory state*. The set of all possible sensory states forms the *sensory space* $\mathcal{S}$.

We can finally define the *locally situated state* of the agent

$$\lambda = (s, \hat{x}), \qquad \lambda \in \Lambda$$

where Λ is the *locally situated state space*, and $\hat{x} \subset x$ are the relevant "internal" state variables of the agent that are independent from information contained in the sensory state s. At most, $\hat{x}$ is the position- and orientation-independent portion of x, but often smaller.[3]

[2]We construe e to contain enough information about the environment's geometry that, together with x, it is sufficient to decide a collision check. In dynamic environments e is a function of time.

[3]For some agents it is useful to post-process $\hat{x}$, such that $\lambda = (s, f(\hat{x}))$, in order to reduce its dimensionality or make it more amenable to learning.

IV. Discovery & modeling of local viability

The second component of our approach entails building up a large pool of experience. Specifically, we are interested in collecting samples of locally situated states λ which are known to be viable, so that we can then derive, through suitable machine learning methods, a viability classifier Ω_v, or *oracle*, such that

$$\Omega_v(\lambda) : \Lambda \rightarrow \{\texttt{viable}, \texttt{nonviable}\}$$

This oracle is then used to guide planning effort.

Viability data can be collected from prior runs, or can be provided by an external source. Naturally, if an adequate external model of the agent's viability is available, one can completely forego this accumulation stage and use the model directly; unfortunately most nontrivial dynamical systems do not have easily obtainable analytical viability kernels, nor prior empirically derived ones, thus necessitating this step.

It's worth noting that the oracle will have two sources of prediction error: modeling error, and limitations in the sensors' ability to disambiguate states. The first is a result of either insufficient training samples or poor choice in training procedure parameters, and is easily corrected. The second type of error is inherent in the collected data, and much harder to fix. The problem stems from the agent's set of sensors being unable to disambiguate between some pairs of system states, ${x^+}_1$ and ${x^+}_2$, where one is viable and the other nonviable, and as a result mapping them both to the same locally situated state λ. Since the oracle will always return the same label for λ, whether it is derived from ${x^+}_1$ or ${x^+}_2$, then one of these system states will always be misclassified. Section VIII discusses further these errors and their consequences.

V. Exploiting local viability

The final component of our approach is the exploitation of the acquired experience. The key idea is that, since by definition it is impossible for a nonviable state to lead to a viable one, then exploring nonviable states is wasteful since they cannot help to reach the goal.[4] By avoiding exploration of such states we can thus expedite the planning process.

Given a trained viability oracle, it is trivial to instrument an arbitrary planner for viability filtering: one merely replaces the default call to a collision or failure checking routine, named `is_collision()` here, with a call to `is_nonviable()`:

1: **function** IS_NONVIABLE(x^+)
2:     **if** `is_collision(`x^+`)` **then**
3:         **return** `True`
4:     $s \leftarrow \sigma_1(x^+),\ \sigma_2(x^+),\ \ldots$
5:     $\hat{x} \leftarrow$`extract_internal_state(`x^+`)`
6:     $\lambda \leftarrow (s, \hat{x})$
7:     **return** $\neg\ \Omega_v(\lambda)$

[4]This assumes that x_{goal} is viable; the special case where x_{goal} is nonviable is discussed later on.

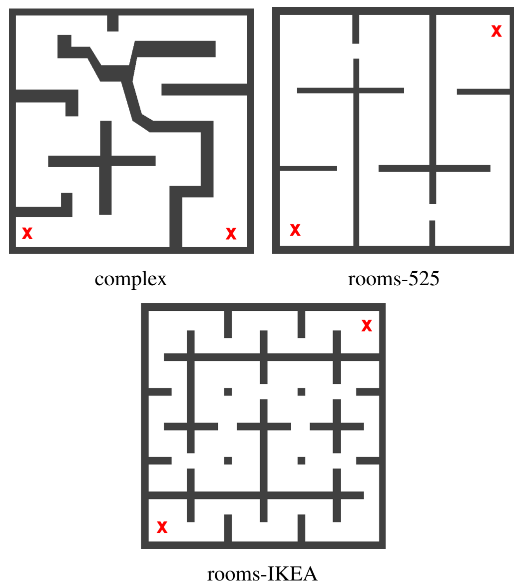

Fig. 3. Examples of problem environments and queries tested. The queries require the agent to traverse from the X on the left to the one on the right in each environment.

`is_nonviable()` still consults `is_collision()` since usually the viability model will not be perfect, and thus if used unaided, could occasionally allow an in-collision state.

VI. Implementation details

To validate the above ideas we have outfitted the RRT-Blossom [1] planner with viability filtering, and tested it on three agents in a number of environments (see Figures 1, 3).

A. Sensing & situation encoding

This section describes only the agent's forward-time sensors; for the reverse-time sensors (i.e., those used in x_{goal} trees, as explained later in VI-C) the logical analogues are used (see dotted lines in Figure 1). In the following, p refers to the subject's 2D position in the environment, while $\mathcal{U}$ refers to the subject's control input space, the set of discrete control actions allowed.

The "inertial point" agent is a point-mass with four constant-force thrusters mounted in the four cardinal directions. During operation the agent is required to have at all times one, and only one of these thrusters on. The agent has upper and lower bounds on its velocity.[5] The subject's state, control actions, sensory state, and locally situated state are:

$$x = (p, p') \qquad s = \sigma_{p'}$$
$$\mathcal{U} = \{\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}\} \qquad \lambda = (s, \|p'\|)$$

where $\sigma_{p'}$ measures the distance from agent to the nearest obstacle along the velocity vector p'.

The non-holonomic car used is very similar to that in [1], but less maneuverable, with a smaller maximum steering

[5]A small lower bound was found useful since otherwise most exploration occurs at near-zero velocities due to the ease with which antagonistic thruster pairs can cancel out each other's progress.

angle ψ_{max}. The relevant variables are:

$$x = (p, \theta) \qquad s = (\sigma_{L_{wh}}, \sigma_F, \sigma_{R_{wh}})$$
$$\mathcal{U} = \{-\psi_{max},\ 0,\ \psi_{max}\} \qquad \lambda = s$$

where θ is the car's orientation, σ_F is a forward-facing rangefinder, while $\sigma_{L_{wh}}$ and $\sigma_{R_{wh}}$ are the left and right "whiskers". A *whisker* returns the distance to the environment along a particular path. It can be useful to consider curved whisker-paths for robots when this reflects the nature of their motion. In practice, we approximate such curved paths using a small set ($n = 8$) of straight line segments for computational efficiency. The car's whiskers correspond to the two extremal steering actions applied for the duration of a 180° turn, as shown in Figure 1.

The last and most complex subject is the dynamic bike model from [1], with the following parametrization:

$$x = (p, \theta, \phi, \phi') \qquad s = (\sigma_L, \sigma_F, \sigma_R) \qquad \lambda = (s, \phi, \phi')$$
$$\mathcal{U} = \{-\psi_{max}, -\tfrac{\psi_{max}}{2},\ 0,\ \tfrac{\psi_{max}}{2},\ \psi_{max}\}$$

where θ is the bike's orientation, ϕ its lean angle, σ_F is again the forward-facing rangefinder, while σ_L and σ_R are rangefinders deflected off-center by 30°, left and right, respectively.

B. Discovery & modeling

The preferred way to accumulate viability samples is through "bootstrapping", whereby the planner progressively builds up Ω_v from scratch using self-observation, since this makes the planner self-contained. However, bootstrapping is currently problematic, as discussed in section VIII-C; instead we collect samples of viable states from very long random-walk trajectories, created by applying random control actions at each time step, and backtracking upon failure.

The above method yields only viable samples since to prove a state is viable it is sufficient to show a single viable outbound trajectory, yet to prove nonviability one must show that *all* possible trajectories out of the state end in failure. Alas, even proving the former requires a concession since a viable trajectory is one that can continue indefinitely, which is not always easy to show. We thus adopt a time horizon, t_h, in our viability assessments (we use $t_h = 10s$); that is, any state which supports a collision-free trajectory longer than this duration is deemed viable. This assumes that the failure modes of the subject are confined to durations shorter than t_h; in our experience, any error introduced with this approximation (i.e., contamination of the experience pool with nonviable states) is insignificant when compared to other sources of error in our approach. In this way then, any search tree or trajectory in the training datasets is readily converted to a collection of sample viable states by simply discarding t_h worth of motion from their tail ends.

Once a sufficient number of samples have been collected, the oracle can be trained. Since the collected samples all belong to the same class (i.e., "viable" set), this is naturally a one-class classification problem. We use the one-class classifier in the `libSVM` [17] library. The SVM learning process is straight-forward: prior to learning, the training samples $\lambda \in \Lambda$ are first column-standardized[6], and then additionally scaled further in some of the dimensions to give those features more "resolution". Radial Basis Function (RBF) is used for the kernels, with $\gamma \in \{0.5, 1, 2\}$, while $\nu \in \{0.005, 0.01\}$.

C. Dual-tree considerations

In dual-tree planners, the x_{goal} tree is usually built using reverse-time simulation of system dynamics. Such trees require some obvious modifications to our filtering approach. Since the subject generally moves "backwards" in reverse-time, it is necessary to flip the sensor orientation front-to-back; see dotted lines in Figure 1. This further implies that reverse-time trees need to build and use their own oracle, $\Omega_{v_{rev}}$, based on sensory data from this "flipped" sensor set. The resulting oracle models viability in reverse-time, which is equivalent to modeling reachability for a forward-simulation setting (see Figure 2), thus requiring that training trajectories be t_h-trimmed from the front rather than the tail.

A nonviable x_{goal} poses a problem for single-tree planners because the viability filtering component will directly deter the planner from completing the last leg of the solution. Fortunately the dual-tree approach completely sidesteps this problem because a solution requires only a tree-tree connection, not a tree-node one. That is, the x_{init} tree, T_{init}, no longer needs to reach x_{goal}; it only needs to reach any node in the x_{goal} tree, T_{goal}, a much larger target. Similarly, x_{init} does not have to lie in reachable space since T_{goal} needs only to reach any node in T_{init}. The only consequence of this is that the two trees may only meet in space which is both, viable and reachable, but this does not seem to have any noteworthy repercussions.

VII. Results

The testing platform was implemented in Python 2.4, on a Pentium IV 2.4GHz Linux machine (kernel 2.6). The base planning algorithm used was the dual-tree RRT-Blossom presented in [1]. To partially offset the slower speed of an interpreted language a number of optimized modules were used: "psyco", the C-implemented libSVM library (through the accompanying Python bindings), "scipy", and other C-implemented modules of mathematical nature. Collision checking was done using a naive Python implementation.

Table I summarizes the numerical findings. The labels "CT", "Blossom", and "BlossomVF" refer respectively to RRT-CT [11], RRT-Blossom [1] and our present algorithm, RRT-Blossom with Viability Filtering. The values are averages over 20 runs for RRT-Blossom and RRT-Blossom/VF, and over 10 runs for RRT-CT. Figure 4 illustrates the effect that the viability filtering has on tree structure.

It is important to note that each agent uses the same viability and reachability models for all three environments. These models were all trained on data from the "complex" environment, which largely explains why the gains in that environment are the most dramatic (shaded row in Table I).

[6] $\lambda_i \leftarrow (\lambda_i - \mu_i)/\sigma_i$, where λ_i is the i'th coordinate of vector λ, while μ_i and σ_i are the corresponding mean and standard deviation, respectively.

TABLE I
PERFORMANCE COMPARISON

		INERTIAL POINT			CAR			BIKE		
environment	**algorithm**	**time (s)**	**# iter.**	**# nodes**	**time (s)**	**# iter.**	**# nodes**	**time (s)**	**# iter.**	**# nodes**
complex	CT	1448.4	20,064.0	21,468.5	301.8	9408.2	9969.4	1088.7	29,775.6	17,070.5
	Blossom	1033.6	10,033.7	12,007.4	186.8	6187.0	6439.3	72.4	4137.1	4019.1
	BlossomVF	**171.1**	2401.0	4208.9	**12.2**	478.7	720.9	**15.9**	651.3	805.5
rooms-525	CT	488.5	14,157.2	12,088.3	204.3	6404.2	7892.0	943.4	23,521.1	14,245.4
	Blossom	1287.2	10,027.3	13,711.2	1428.5	19,308.5	19,895.9	193.3	6904.0	6695.1
	BlossomVF	**326.0**	3232.5	5682.9	**63.4**	1954.5	2503.2	**89.6**	2669.8	2773.4
rooms-IKEA	CT	1403.7	26,018.4	20,114.4	1142.6	22,044.1	20,086.9	409.8	19,146.4	9765.1
	Blossom	1062.4	9839.0	12,241.0	1425.5	19,040.0	19,605.0	105.7	4910.3	4750.3
	BlossomVF	**481.2**	4578.1	7113.6	**685.0**	7144.6	9617.9	**40.8**	1066.1	1291.8

At the same time, it is encouraging to see that the models are still effective when applied to other environments. The diminished gains in "non-native" environments are a consequence of the sensors' perceptual limits introducing environment-specific artifacts into the learned model, and such artifacts are not generally transferrable.

VIII. DISCUSSION

A. Choice of sensors

The gains of viability filtering are directly tied to the net balance of work saved (skipping exploration of futile branches) minus extra work incurred (computing sensor values and consulting the oracle). It is thus important to find sensors that are relatively cheap to compute, yet at the same time ones that capture the viability-relevant aspects of the agent state particularly well. Excessive emphasis on reduced computational cost can be counterproductive, however: for many dynamical systems the increased computational cost of more perceptive sensors is far outweighed by the resultant heavier pruning of the search space. For example, even though it is far cheaper to compute the sensory state of a car outfitted with three linear rangefinders, rather than the whiskers, this produces poorer results, despite the much reduced sensor computation time.

B. Consequences of oracle error

Modeling error in the viability oracle can be either underinclusive (false negatives; viable states labeled as nonviable) or overinclusive (false positives; nonviable states labeled as viable). An underinclusive model restricts planner exploration more than it should. This results in a smaller search space, and consequently shorter planning times, but it is detrimental in highly constrained environments since essential bottlenecks are easily made impassable when misclassified as nonviable. Overinclusive models, on the other hand, diminish the amount of filtering of the search space, causing the planner to gradually regress into the host planning algorithm (i.e., the form without filtering) as this error increases. In general, the amount of viability filtering is a function of model error, and spans a continuous spectrum as illustrated in Figure 5.

C. Scarcity of samples & bootstrapping

There are two key issues with bootstrapping. The primary is that viability filtering directly prevents the discovery of novel viable samples: any such sample would be necessarily misclassified by the current oracle, and thus filtered from exploration. In short, a planner with full viability filtering cannot extend its viability model using self-observation. Secondly, even if this were not so, the initial dearth of samples during bootstrapping would lead to heavily underinclusive models, which would in turn lead to frequent inability to find solutions to queries, thus again restricting the availability of fresh training data.

Both problems can likely be overcome by stochastically phasing-in the viability filtering. That is, filtering could be applied only to iterations in which $r > \Phi$, where $r \in [0, 1]$ is a random variable, and $\Phi \in [0, 1]$ is a phase-in parameter that increases as the viability model fills out. The model's "fullness" could perhaps be gauged by the inverse

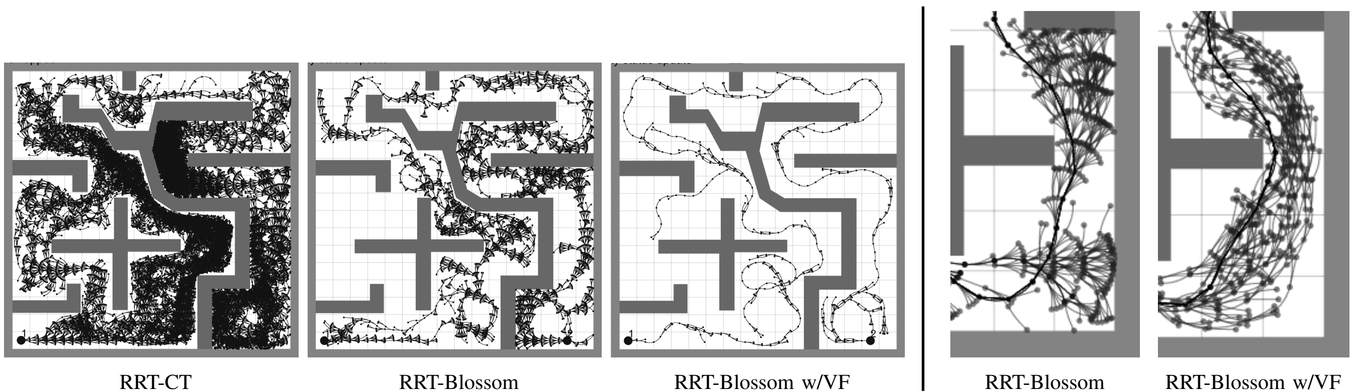

Fig. 4. The effect of viability filtering on tree structure and density (bike results shown). On the left are the three compared algorithms; the filtering planner (rightmost) achieves a solution with noticeably less effort. On the right is a magnified view of tree structures for the bike. The filtering planner (on right) conspicuously avoids probing the corners and instead relies almost exclusively on viable trajectories.

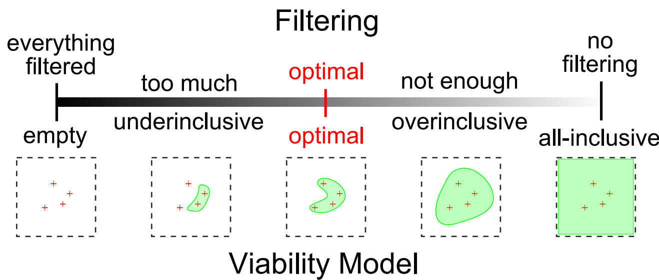

Fig. 5. Amount of viability filtering as a function of model error.

of the rate at which novel samples are encountered. The general problem is similar to that of exploration-exploitation tradeoffs encountered in reinforcement learning.

It is hard to characterize the number of samples required to adequately capture the viable region. In general, the model grows in proportion to the rate at which novel samples are encountered, ones that lie outside current model bounds, and the degree of their novelty. A practical characterization might be to claim an adequate model when this rate of novel samples fall below some predetermined, sensible threshold.[7]

D. Oracle transferability

As the results show, a viability model can be effective in environments other than the one that was used in training. How well a model transfers is dependent on the degree to which the environments are similar in character and structure. For example, a model trained in a constrained environment will do poorly in wide open areas (and vice versa) since the model will be mostly limited to Λ's origin, whereas the new environment will generally keep agent operation in more distant, unexplored regions of Λ. The obvious remedy to counter such over-specialization is to train the oracles on samples from a variety of dissimilar environments. Initial experiments (conducted only on the car so far) have yielded good results.

E. Expected reduction in planning effort

Our approach hinges on preventing the exploration of branches which are very unlikely to lead to a solution, hence the reduction in planning effort is proportional to the prevalence of such branches, which are usually the result of the interaction of system dynamics with imposed constraints (e.g., environment geometry). Such branches will typically be more numerous when the agent is generally unstable, has a very limited range of motion, or the environment is relatively constraining. The speedup will further depend on how conservative a viability model the oracle derives from training data (see §VIII-B). Finally, results will deteriorate if the target environment significantly differs in structure from those used in training, or when the sensors used do not sufficiently capture the local context of the agent.

[7]This is still not very general since it assumes that the training samples are drawn uniformly from Λ; this is usually not the case.

IX. Conclusion

This paper proposes the use of locally situated state information as a means to give motion planners "sight", which then aids in the detecting and learning nonviable scenarios. This learned viability data can then be exploited to expedite planning by barring the planner from wasting effort on exploring nonviable regions of space. Results are shown for three types of agents and demonstrate significant speedups and generalization across environments.

Many interesting research directions remain. Annotating the viable state samples with their corresponding control actions, as found in the training data, could lead to more powerful models. The oracle could adjudicate viability based on λ *histories*, rather than just a single situated state, thus opening up many possibilities. Combined with the collected control action data, this may lead to automatic identification of macroscopic motion primitives for the agent using pattern matching and clustering of the λ histories. This could also substantially compensate for perceptual limitations of the agent's sensors. The discovery of effective sensors could perhaps be automated. Finally, bootstrapping of the viability oracle and the compositing of training data from multiple environments could be further explored.

References

[1] M. Kalisiak and M. van de Panne, "RRT-Blossom: RRT with a local flood-fill behavior," in *IEEE Int. Conf. Robotics & Automation*, 2006.
[2] J.-C. Latombe, *Robot Motion Planning*. Kluwer Academic Press, 1991.
[3] S. M. LaValle, *Planning Algorithms*. Cambridge, 2006.
[4] T. L. Dean and M. P. Wellman, *Planning and Control*. Morgan Kauffman, 1991.
[5] J. Barraquand and J.-C. Latombe, "Robot motion planning: A distributed representation approach," *The International Journal of Robotics Research*, vol. 10(6), pp. 628–649, 1991.
[6] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Computer Science Dept., Iowa State University, technical report TR 98-11, 1998.
[7] S. M. LaValle and J. J. Kuffner, Jr., "Rapidly-exploring random trees: Progress and prospects," in *Workshop on Algorithmic Foundations of Robotics*, 2000.
[8] M. H. Overmars and P. Svestka, "A probabilistic learning approach to motion planning," in *Workshop on the Algorithmic Foundations of Robotics*, 1995.
[9] S. M. LaValle and J. J. Kuffner, Jr., "Randomized kinodynamic planning," in *IEEE Int. Conf. Robotics & Automation*, 1999.
[10] B. R. Donald, P. G. Xavier, J. F. Canny, and J. H. Reif, "Kinodynamic motion planning," *Journal of the ACM*, vol. 40, no. 5, pp. 1048–1066, 1993.
[11] P. Cheng and S. M. LaValle, "Reducing metric sensitivity in randomized trajectory design," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2001.
[12] S. Caselli and M. Reggiani, "ERPP: An experience-based randomized path planner," in *IEEE Int. Conf. Robotics & Automation*, 2000, pp. 1002–1008.
[13] M. Kalisiak and M. van de Panne, "Approximate safety enforcement using computed viability envelopes," in *IEEE Int. Conf. Robotics & Automation*, vol. 5, 2004, pp. 4289–4294.
[14] J.-P. Aubin, *Viability Theory*, ser. Systems & Control: Foundations & Applications, C. I. Byrnes, Ed. Birkhäuser, 1991.
[15] I. Mitchell, "Application of level set methods to control and reachability problems in continuous and hybrid systems," Ph.D. dissertation, Stanford, 2002.
[16] L. Chapel and G. Deffuant, "SVM viability controller active learning," *Kernel machines for reinforcement learning workshop*, 2006.
[17] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001, http://www.csie.ntu.edu.tw/~cjlin/libsvm,.