2008 IEEE/RSJ International Conference on Intelligent Robots and Systems
Acropolis Convention Center
Nice, France, Sept, 22-26, 2008

Replanning: A powerful planning strategy for hard kinodynamic problems

Konstantinos I. Tsianos and Lydia E. Kavraki

Abstract— **A series of kinodynamic sampling-based planners have appeared over the last decade to deal with high dimensional problems for robots with realistic motion constraints. Yet, offline sampling-based planners only work in static and known environments, suffer from unbounded memory requirements and the produced paths tend to contain a lot of unnecessary maneuvers. This paper describes an online replanning algorithm which is flexible and extensible. Our results show that using a sampling-based planner in a loop, we can guide the robot to its goal using a low dimensional navigation function. We obtain higher success rates and shorter solution paths in a series of problems using only bounded memory.**

I. Introduction

Motion is essential for any autonomous robot to be able to perform useful tasks. The simplest case is path planning, where a collision-free kinematic path that takes a robot from an initial to a final configuration is sought. A more interesting class of problems is that of *kinodynamic* motion planning. These problems are harder because they take into account the physical limitations of a robot by including acceleration constraints and non-integrable velocity constraints. The complexity of motion planning even for the kinematic case, renders complete algorithm intractable but for the simplest cases [14]. A breakthrough in the area was achieved in the last two decades with the development of *sampling-based planners* that tradeoff completeness for computational efficiency [5]. With the appearance of sampling-based planners, researchers became more interested in problems with many degrees of freedom and problems that are more realistic. Realism in motion is driven both from robotic applications but also from other areas, like the computer game industry.

So far, there have been some important steps towards solving hard and realistic problems using (kinodynamic) sampling-based planners [17]. In [11] the kinodynamic version of Rapidly exploring Random Tree (RRT) can plan for hovercrafts and satellites. Other encouraging results are found in [3] and the Expansive Space Tree (EST) [4]. Path Directed Subdivision Tree (PDST) [8], [10] is a recent algorithm that introduced the idea of using a physics simulator with the sampling-based planner. In this way it is possible to incorporate realistic physical constraints such as gravity and friction.

Despite all the recent successes, there are still a lot of issues that need to be addressed. Some are related to performance. In most cases, (kinodynamic) sampling-based planners are designed to be and are generally thought of as offline planners. The problems studied in our experiments section expose some important concerns related to offline planning. For high dimensional problems, planners such as RRT and PDST tend to require large amounts of time and memory to compute a solution. This is more pronounced in kinodynamic problems where the robot has a much more limited set of allowable motions. Moreover, from our experience even when the solution is found, the paths to the goal tend to be very long and contain a lot of unnecessary maneuvers. Finally, an offline algorithm needs complete knowledge of the workspace and approaches the motion planning problem in isolation. There is no room for interfacing the planner with other modules such as task planning, learning and sensing.

This work was supported by NSF IIS 0713623, CNS 0454333, and CNS 0421109. The authors are with the Computer Science Department, Rice University, Houston, TX,77005, USA konstantinos,kavraki@rice.edu

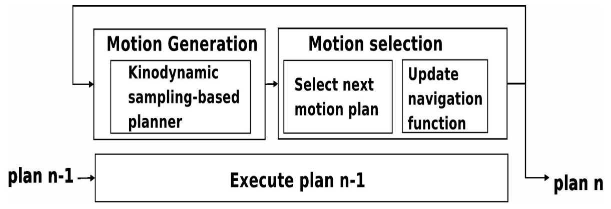


Fig. 1. Replanning: During replanning period n, the algorithm computes the next motion plan while the robot executes the plan computed in the previous replanning period.

This paper follows a sampling-based approach but proposes a new way of using the planner. Our algorithmic framework tries to utilize the power of modern sampling-based planners while dealing with the issues mentioned above. The underlying idea is to use a kinodynamic sampling-based planner in a closed loop and call it periodically (see Fig. 1). Planning is an online incremental process where the robot computes its future plan as it moves. Every plan hopefully brings the robot closer to its goal and the total solution is found after a series of consecutive replanning steps. The tradeoffs when using such an approach will be discussed later in the paper.

The idea of replanning is well established in the area of artificial intelligence [6], [12] and there have been some extensions for sampling-based planners both for kinematic [2] and kinodynamic problems [1]. All these papers justify the use of replanning by the fact that in unknown environments and in environments with moving obstacles offline planning is not applicable. Although our ultimate goal is to address such problems, this paper is an intermediate step and contains a detailed study of the benefits that replanning can yield in hard motion planning problems, even for cases where offline planning is applicable. For the problems we examined, we show experimentally that the replanning versions of state-of-the-art sampling-based planners like RRT and PDST, are more effective than the offline versions, while using only bounded amounts of memory. Moreover, even in cases where the offline planners compute the solution faster, the replanning algorithms produce shorter paths and the robot takes much less time to actually move to its goal location.

The basic modules of our algorithm are described in section II. Section III contains our simulated experiments. The concluding section IV discusses the ongoing and future extentions of this work.

II. Algorithm

The overall algorithm is a combination of modules with distinct responsibilities. A block diagram is shown in Fig. 1. The planning process is broken down to replanning periods $P_0, P_1, P_2 \dots$ of equal duration. There are two major steps within each replanning period; *Motion Generation* and *Motion Selection*. Their purpose is to compute the robot's next motion. Computation takes place while the actual robot is in motion, executing the plan computed in the previous replanning period. During replanning period P_n the robot executes the motion plan that was computed in period P_{n-1} and computes the plan that will be executed in period P_{n+1}. This is reasonable under the assumption that there is no motion uncertainty and thus the state of the robot after executing a motion can be computed in advance.

The rest of this section is devoted to describing the modules within a replanning period in more detail.

A. Motion Generation

Given the state at which the robot will be after executing the current plan, a motion generation phase is responsible for computing a set of possible future motions. Those motions can be thought of as possible future actions for the robot. This paper focuses on the use of a kinodynamic sampling-based planner.

A typical sampling-based kinodynamic planner proceeds as follows. First a sample is selected. Then, a set of random controls are sampled. Finally, those controls are applied on a simulated model of the robot. Starting from the state indicated by the selected sample, a new sample is generated by integrating the robot model forward in time. By repeating those steps, the planner produces a kinodynamic tree of samples. With the information stored on the samples, each path down the tree represents a possible future motion. By extracting all the paths of the tree that describe motions of durations of at least one replanning period we obtain the desired discrete set of feasible motions that will be passed on to the Motion Selection module.

The kinodynamic planner is called once in every a replanning period and has only limited time to run. For this reason, we allow the kinodynamic planner to search only a small part of the space around the robot's state. Contrary to offline kinodynamic planning, this is acceptable here since each motion is not expected to have long term effects; our replanning periods typically need to last from 0.5 to 1 second or less[1]. Experiments examining the effect of the varying the replanning period duration are given in section III-E below. Within its time budget, it is important that the kinodynamic planner achieves a fairly good local coverage of the space around the robot to provide motion options in all directions.

Motion Generation is one of the most computationally intensive steps and it is desirable to have an efficient implementation. An important step that can improve performance of the kinodynamic planner in a replanning framework is tree retainment. When the real robot executes a motion, it is moving down on a specific path of the kinodynamic tree. Consequently, the subtree that follows that path is still usable and can be retained to be used as a starting point for the kinodynamic planner in the next replanning period. In systems with complicated dynamics this allows the planner to have some available motion options for the imminent future even when sampling fails a lot to extend the tree.

B. Motion Selection

Given a set of feasible motions, this module is responsible for making the best motion selection that will take the robot closer to its goal. In order to make this selection, we use a navigation function to evaluate motions.

1) Navigation function: The Motion Selection module needs to have a sense of direction and the ability to quickly answer what is the best possible motion towards the goal from any state. A navigation function with a single global minimum at the goal that captures the true distance from any state to the goal would be a perfect candidate [7], [15]. Yet, designing such navigation functions can be a hard and computationally expensive task for the kinodynamic problems we are interested in. We resort to a discrete navigation function on the 2D or 3D workspace, computed with a wavefront propagation algorithm. The cells that are covered by an obstacle take a value of infinity and don't participate in the computation. The rest of the cells get an integer value representing the distance (in number of cells) to the goal. The navigation function encodes all the possible paths from any location to the goal. Fig. 2 shows an example for a robot on a plane that needs to go from the bottom left corner to the top left. In Fig. 2a, the darker the color, the farther that cell is from the goal. As a result, the preferred route is $R1$ through the narrow passage. There is also route $R2$ which initially appears longer. In Fig. 2b the navigation function is drawn as a 3D surface with the height at each point being the value of the corresponding cell.

Given the navigation function, we evaluate all the candidate motion plans produced by Motion Generation. For each motion plan we find inside which cell that motion ends. The motion plan is assigned the value of that cell. Motion Selection chooses the motion with the minimum value since this will take the robot closer to its goal.

The caveat of computing the navigation function on the workspace is that it can be misleading. Route $R1$ in Fig. 2a

[1]One can also imagine scenarios of an unknown or unpredictable environment where it is not safe to search outside the sensing range of the robot, but this case is beyond the scope of this paper.

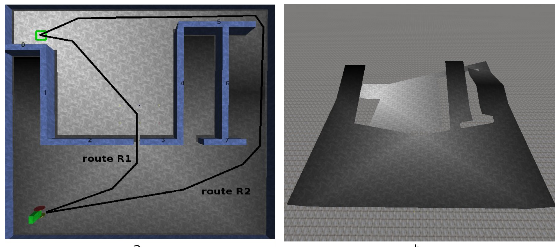


Fig. 2. a. Navigation function on a 2D workspace. Darker means further away. b. A 3D illustration of the navigations surface.

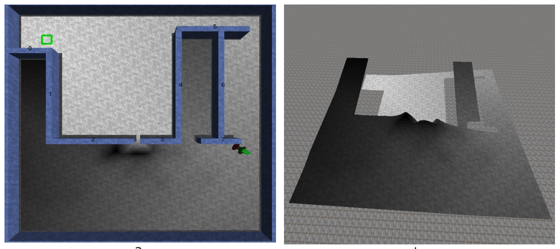


Fig. 3. a. After some updates, the penalization around the narrow passage of route $R1$ makes this route less preferable b. A 3D illustration. Now route $R2$ is a better option.

leads to the goal through a narrow passage. A point robot can follow the series of cells with decreasing values indicated by $R1$ to reach the goal. A real robot can easily get stuck in front of a narrow passage through which it cannot fit, even if there is a solution via another route (route $R2$ in Fig. 2a). To resolve this issue the navigation function is updated as the robot moves.

2) Updating the navigation function: The intuition is that a robot advancing to its goal should not go through the same area many times. If that happens, this is an indication that the robot is moving in circles and/or has difficulty moving in the direction suggested by the navigation function. It also means that it may be worth exploring other alternative routes to the goal before spending more time in this region - possibly a hard narrow passage. To make such a decision, one needs to detect the lack of progress. We capture it by keeping a *penalty* value for each cell, representing how much it has been visited. The navigation function computation is shown in algorithm 1. The penalty values for all cells are initially zero.

Algorithm 1 NavigationFunction(c_{goal})

```
1: $c_{goal} = 0$
2: Q.push($c_{goal}$)
3: while ($Q \neq \emptyset$) do
4:     c = Q.pop()
5:     for all $c' \in Neighborhood(c)$ do
6:         if $c' \neq Obstacle$ and $c'$ has no value then
7:             $c'.value = c.value + c'.penalty + 1$
```

If a region of cells is penalized enough, the navigation function landscape changes and the robot naturally tries to go another way. Fig. 3a shows the effect of the updates. The robot tried repeatedly to follow route $R1$ without success. After spending some time in front of the narrow passage, the navigation function changed due to penalization. This is reflected by the dark cells around the narrow passage of $R1$ (Fig. 3a). As a result, the robot was lead to take route $R2$ (In Fig. 3b, route $R2$ is now lower than $R1$). We emphasize the fact that the penalization process does not introduce local minima to the navigation function as this would render the whole approach problematic.

Probabilistic completeness: Most offline sampling-based planners are *probabilistically complete*. If a solution exists the planner will find it given enough time. In a replanning framework the kinodynamic planner runs under strict time limitations, and probabilistic completeness is lost. Although in our experiments we have never observed our algorithm getting stuck in one part of the space, it is an open question whether our updating technique can effectively restore probabilistic completeness. Note also that probabilistic completeness is only meaningful in static environments. Replanning is eventually intended for planning in changing and partially known environments where probabilistic completeness may need to be redefined as a concept.

III. Experiments

A. Experimental setup

Robots: This section presents a series of experiments on two autonomous robots; a segway and a blimp. Both robots present significant difficulties to planners because they are high dimensional, underactuated and have kinodynamic constraints. The segway moves on a plane and its state can be fully described by seven parameters $(x, y, \theta, \alpha, \dot{\alpha}, v, \dot{\theta})$. x and y provide the location and θ the orientation. α provides the segway's tilting angle and the last three parameters are velocities. The segway is controlled only by two torques applied on its two wheels. The blimp is a model for a balloon and moves in a 3D workspace. Its states are described by parameters $(x, y, z, \theta, V_{fwdx}, V_{fwdy}, V_z, \omega)$. Its location is given by (x, y, z) and its orientation is θ. Its linear velocity has two components V_{fwdx} and V_{fwdy} that are parallel to the ground and one vertical V_z. There is also a rotational velocity ω describing the rate of change of orientation. A blimp has three controls. One force along the axis $(cos(\theta), sin(\theta), 0)$, a second vertical force along $(0, 0, 1)$ and a torque around $(0, 0, 1)$ [2]. All forces and torques for both models are bounded and the robots are affected by gravity and ground friction (for the segway). To simulate the behavior of the robots the ODE open source physics simulator was used [16].

Algorithms: For our experiments, we used two kinodynamic planners. A variant of the PDST algorithm and RRT. In the rest of this Section, *PDST* and *RRT* stand for the offline planners, while *RPDST* and *PRRT* are the replanning versions. *RRT* is described in [11]. The bias used, was 7%

[2]For an extensive illustration of the segway dynamic model see [13]. The blimp model is given in more detail in [9].

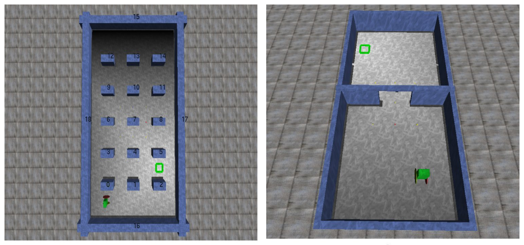


Fig. 4. a. The *block* environment, b. The *door* environment.

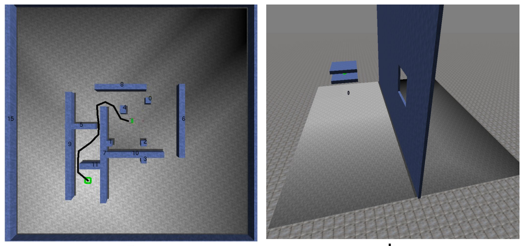


Fig. 5. a. The indoor *building* environment, b. The environment for the blimp

for the offline version and 3% for the replanning version. The distance metrics were hand-tuned weighted Euclidean distances for both the segway and the blimp. For the segway the weights were $w_x = w_y = 0.35, w_\theta = 0.1, w_\alpha = 0.1, w_{\dot{\alpha}} = w_v = w_{\dot{\theta}} = 0.033$. For the blimp the weights were $w_x = w_y = w_z = 0.32, w_\theta = 0.0009, w_{velocities} = 0.009$.

The basic PDST implementation is given in [10]. The algorithm follows the steps described in section II-A. *PDST* uses a space subdivision scheme to bias the search towards empty unexplored areas and every sample knows the volume of the part of the space it lies in. To allow for backtracking, each sample also has a priority value which increases exponentially everytime the sample is selected. Each sample is assigned a score computed as $score = \frac{priority}{Volume}$. In every selection step the sample with the minimum score is selected. It is clear that basic PDST has no biasing towards the goal as RRT does. For fairness of comparison, we have added a simple bias using the navigation function. In our implementation sample scores are computed as $N * \frac{priority}{Volume}$ where N is the distance estimation given by the navigation function. In this way, samples that are closer to the goal (lower N) have lower scores and have more chances of being selected for future propagation attempts. To plan for the robots described above, subdivision was done on the x, y, θ, α dimensions for the segway and on x, y, z, θ for the blimp.

Finally, for the replanning versions in all our experiments the duration of the replanning period was set to 0.5 sec.

Environments: The environments for our experiments were designed to test the above algorithms in a variety of narrow passages and cases of a misleading navigation function. The *block* environment (Fig. 4a) is a structured and symmetric environment with multiple ways to reach the goal. The *door* environment (Fig. 4b) contains an interesting and slightly unusual narrow passage. The segway needs to go under a door. Given the door's height, this is only possible if the segway is tilted enough, which in turn requires the segway to be moving fast enough. The third segway environment is the *building* (Fig. 5a). The shortest route to the goal leads through a passage that is too narrow for the segway. The goal is reachable via two longer routes. The blimp was tested on one environment (Fig. 5b). The blimp has to fly through a window and then hover between two parallel plates to reach its goal.

Setup: All experiments were performed on one core of an desktop machine with an Intel Core 2 duo processor and 4 gigabytes of memory. All experiments have a time limit of 10 minutes, and are averages over 50 runs.

B. PDST vs RRT

First, we are interested in comparing performance of the offline planners. Since RRT is a well-known standardized algorithm, this will help us confirm that PDST is also powerful state-of-the-art planner. The comparison was performed on the *building* and *door* environments. In *building*, *PDST* solved the problem 95% of the time and took on average 40.2 sec. *RRT* succeeded only 12% of the time and runtimes averaged to 433.1 sec. In *door* we ran experiments of incremental difficulty by lowering the height of the door. Table I shows the results for four height values. The left column states how tilted the segway must be to be able to go under the door. The segway's maximum tilting angle is 25 degrees. In both environments *RRT* requires more time and fails more often, so *PDST* is indeed a powerful kinodynamic planner. The difference in performance could also be indicating that using the navigation function to bias *PDST* can be very powerful[3].

Difficulty (deg)	PDST		RRT	
Very Easy : $(8.8-25)$	1.5 sec	100%	68.9 sec	99%
Easy : $(12.13-25)$	2.1 sec	100%	87.9 sec	98%
Medium : $(16.26-25)$	16.5 sec	100%	171.53 sec	19%
Hard : $(21-25)$	N/A	0%	N/A	0%
Very Hard : $(23-25)$	N/A	0%	N/A	0%

TABLE I

C. Replanning vs Offline planning

This is an important set of experiments, that exhibit the beneficial effect of replanning. Due to the decreased performance of *RRT*, our experiments are limited to *pdst*, *RPDST* and *RRRT*.

Segway: Of interest is the behavior of the algorithms as the difficulty of a problem increases. In *block* we set five goals that are increasingly far from the starting location. In *door*

[3]Of course, this may not be the true difference between the algorithms, since RRT can be handtuned to have better performance. Note though, that most powerful versions use a bidirectional tree. Since we use a physics simulator and allow the segway to slip, it is not possible to use those versions since they require the ability to integrate backwards in time.

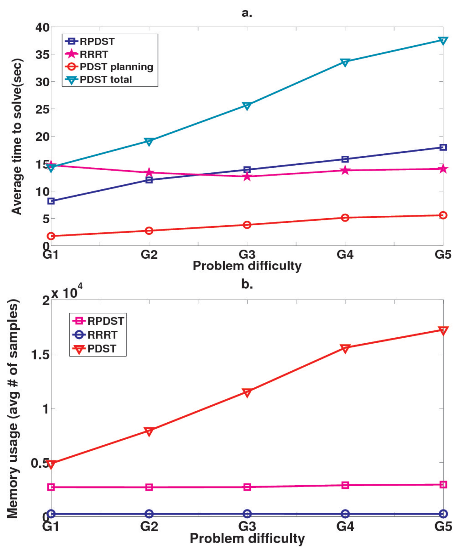


Fig. 6. a. Time to plan and solve increasingly hard problems in *block* using *PDST* , *RPDST* and *PRRT* , b. Average memory requirements.

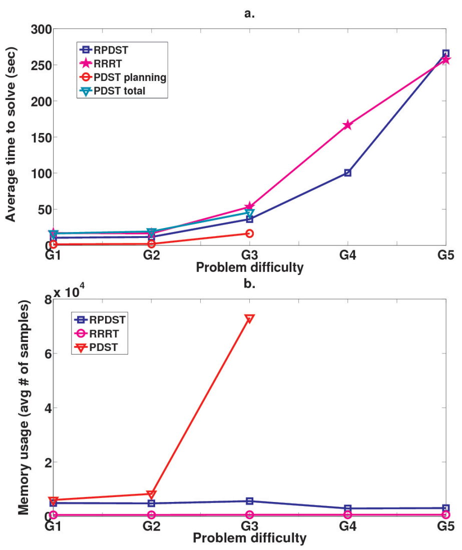


Fig. 7. a. Time to plan and solve increasingly hard problems in *door* using PDST, R-PDST and R-RRT, b. Average memory requirements.

we have five goals by decreasing the height of the door as described in the previous subsection.

Figures 7a and 6a plot the time it takes to solve each problem. *PDST* uses its "global" view of the space to find the solution faster that the replanning algorithms. Faster planning though comes at price in memory as shown in figures 7b and 6b. As the problems get harder, especially in *door*, the memory requirements explode. Replanning finds the solution as concatenation of small partial paths. As expected, the memory requirements are practically constant with *RPDST* being more memory intensive over *PRRT* (by a constant factor). The increasing demand in memory has severe negative effects. In *door*, there is no data for PDST beyond G3 since the planner never solved the problem within the 10 minutes time limit. This is probably because the data structures become very slow when lookups and updates are needed on large numbers of samples. Allowing for more time would eventually lead to memory exhaustion forcing the algorithm to terminate. On the other hand, the replanning algorithms only use bounded memory and can thus run indefinitely.

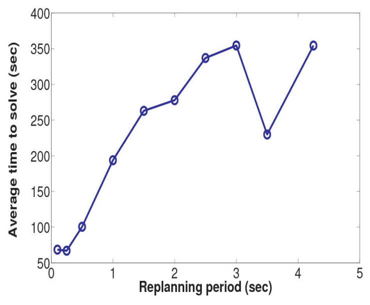
Looking back at the time plots in Figures 7a and 6a an interesting observation can be made. When solving a planning problem, the time needed for the real robot to execute the plan and move to its final location is a factor to consider. For the replanning algorithms this time is the same as the planning time. *PDST* runs offline so we also plot the *total time* which is the planning time plus the time it takes to execute the plan. Notice that the replanning algorithms are doing much better in *block* and equally good in *door* until G3. This means that the paths produced by replanning tend to be shorter. This is a interesting finding. Replanning is in a sense, a divide and conquer greedy strategy. The total problem is broken down into smaller ones, and in each step, the robot executes the best possible motion. Guided by the navigation function, this strategy can yield much shorter paths.

Notice also, that although *RRT* was shown to have significant difficulties in the problems at hand, its replanning version is very competitive, and in some cases even slightly better than *PDST*. This is explained by the fact that *RRT* and *PDST* were compared in results for solving the full problem. When we run *RRT* and *PDST* for the duration of a replanning period, the differences become insignificant.

Blimp: Our results so far, were based on experiments with the segway. The experiments we have for the blimp also confirm our findings. All planners solved the problem on all runs. *PDST* solved the problem at an average of 37.65 sec just for planning and 190.26 sec total when the execution time is added. On the other hand, *RPDST* 's average runtime was 153.02 sec and *PRRT* 's 125.06 sec. Again the offline planner find the solution much faster. Yet, *PDST* required on average 30 times more memory than *RPDST* and about 300 more than *PRRT* . Furthermore, the paths computed by *PDST* are much longer than those of *RPDST* and *PRRT* as indicated by the total times above.

D. Effect of Updating the Navigation Function

Up to now *PRRT* and *RPDST* were tested on problems where the navigation funtion was not misleading. Recall that in *building* the navigation function leads through a passage that is too narrow. As mentioned in Section II, to detect such cases, the navigation function is updated as the robot moves. In this experiment, each cell got a penalty of 0.05 every time it was used. The results are encouraging as both replanning algorithms found the solution in 100% of the attempts. *RPDST* took 47.7 sec in average while *PRRT* took 69.95 sec. To compare, *PDST* 's planning time was reported



RP(sec)	success
0.1	96 %
0.25	100 %
0.5	100 %
1	88 %
1.5	68 %
2	53 %
2.5	44 %
3	40 %
3.5	28 %
4.25	17 %

Fig. 8. Time it takes *RPDST* to solve G4 in *door* for replanning periods of different duration and the success ratios for finding the solution in 10 minutes.

in III-B to be 40.2 sec and the average *total time* was 98.51 sec.

E. Duration of replanning period

Our final set of experiments, investigates the effect of the duration of a replanning period (RP). Notice that if the replanning period is very small the algorithms tend to become more "reactive", while as the replanning period gets longer the algorithms get closer to offline planners. Figure 8 shows the time it takes to solve G4 in *door* for different replanning periods. Interestingly, as the periods get longer, performance deteriotes. The time to solve increases, and the success ratio drops. The results seem to indicate that the replanning period must be set to the shortest possible value, that is still long enough for the motion generation phase to be meaningful.

IV. Conclusions

Recently there has been an increased interest towards high dimensional and more realistic motion planning problems. Some significant progress was made with the use of kinodynamic sampling-based motion planners. Yet, offline kinodynamic planners still face difficulties for some hard and interesting problems. They use significant amounts of time and memory resources and the computed solutions tend to contain a lot of redundant and unecessary motions for the robot. In this paper, we presented a set of problems that expose these issues.

Our proposed solution is to use an online replanning strategy. Using sampling-based kinodynamic planners in a closed loop in combination with a guiding navigation function has allowed us to address the time and memory issues mentioned above. Our experiments with two state-of-the-art sampling-based planners show significant improvements in terms of total runtime and success rates for online replanning against their offline counterparts. Furthermore, the algorithms run using only bounded memory resources. Sampling-based planners are excellent at doing a local search and solving narrow passages but also tend to lose time searching uninteresting parts of the space. Within our framework, the planner uses the navigation function to focus on the critical parts of the space. The ability to backtrack and search alternative possibilities is achieved by constantly updating the navigation function.

This paper is a study of the benefits of replanning in problems with static and known environments. While this may be the limit for offline planners, it is only the starting point for replanning algorithms. Replanning is by default adaptive and captures the fact that knowledge about the robot's environment might change in time. Our current and future work focuses on addressing problems with dynamic and partially known environments, adding a higher level tasks planner in the replanning loop to deal with more complex tasks and augmenting the loop with an estimation module that deals with motion uncertainty.

References

[1] Kostas E. Bekris and Lydia E. Kavraki. Greedy but safe replanning under kinodynamic constraints. In *IEEE International Conference on Robotics and Automation*, pages 704–710, Rome, Italy, April 2007. IEEE press.

[2] Dave Ferguson, Nidhi Kalra, and Anthony Stentz. Replanning with RRTs. In *IEEE International Conference on Robotics and Automation*, pages 1243– 1248, 2006.

[3] Emilio Frazzoli, Munther A. Dahleh, and Eric Feron. Real-time motion planning for agile autonomous vehicles. In *American Control Conference*, volume 1, pages 43–49, 2001.

[4] David Hsu, Roben Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21(3):233–255, 2002.

[5] Lydia E. Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, August 1996.

[6] Sven Koenig and Maxim Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 968– 975, 2002.

[7] Prashanth Konkimalla and Steven M. LaValle. Efficient computation of optimal navigation functions for nonholonomic planning. In *Proceedings of the First Workshop on Robot Motion and Control*, pages 187–192, 1999.

[8] Andrew M. Ladd. *Direct Motion Planning over Simulation of Rigid Body Dynamics with Contact*. PhD thesis, Rice University, Houston, Texas, December 2006.

[9] Andrew. M. Ladd and Lydia. E. Kavraki. Fast tree-based exploration of state space for robots with dynamics. In M. Erdmann, D. Hsu, M. Overmars, and A. F. van der Stappen, editors, *Algorithmic Foundations of Robotics VI*, pages 297–312. Springer, STAR 17, 2005.

[10] Andrew M. Ladd and Lydia E. Kavraki. Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems*, pages 233–241, Boston, MA, June 2005. MIT Press.

[11] Steven M. LaValle and James J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, May 2001.

[12] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[13] Kuaustubb Pathak, Jaume Franch, and Sunil K. Agrawal. Velocity and position control of a wheeled inverted pendulum by partial feedback linearization. In *IEEE Transactions on Robotics*, 2005.

[14] John H. Reif. Complexity of the mover's problem and generalizations. In *IEEE Symposium on Foundations of Computer Science*, pages 421–427, 1979.

[15] Elon Rimon and Daniel E. Koditschek. Exact robot navigation using artificial potential functions. *IEEE Transactions onRobotics and Automation*, 8:501–518, 1992.

[16] Russell Smith. Open dynamics engine. http://www.ode.org. seen September 7, 2007.

[17] Konstantinos I. Tsianos, Ioan A. Şucan, and Lydia E. Kavraki. Sampling-based robot motion planning: Towards realistic applications. *Computer Science Review*, 1(1):2–11, August 2007.