Useful Cycles in Probabilistic Roadmap Graphs

Dennis Nieuwenhuisen   Mark H. Overmars
Institute of Information and Computing Sciences
Utrecht University, the Netherlands
Email: {dennis,markov}@cs.uu.nl

Abstract— Over the last decade, the probabilistic road map method (PRM) has become one of the dominant motion planning techniques. Due to its random nature, the resulting paths tend to be much longer than the optimal path despite the development of numerous smoothing techniques. Also, the path length varies a lot every time the algorithm is executed. In this paper we present a new technique that results in higher quality (shorter) paths with much less variation between the executions. The technique is based on adding useful cycles to the roadmap graph.

I. INTRODUCTION

Automated motion planning has become important in various fields. Originally the problem was mainly studied in robotics, but in the past few years many new applications arose in fields such as animation, computer games, virtual environments, maintenance planning and training in industrial CAD systems. Such applications put new demands on the motion planner.

Over the years, many different approaches to solving motion planning problems have been suggested. See Latombe [18] and the proceedings of the yearly IEEE International Conference on Robotics and Automation (ICRA) or the Workshop on Foundations of Robotics (WAFR).

The probabilistic roadmap planner (PRM) has become a popular technique to solve the motion planning problem. It has been developed at different sites [3], [4], [15], [16], [20], [23]. Globally speaking, the PRM approach samples the configuration space for collision-free positions. These are added as vertices to a roadmap graph. Pairs of promising vertices in the graph are chosen and a simple local planner is used to try to connect such placements with a path. This process is repeated until the graph covers the connectedness of the space.

While over the years many techniques have been proposed to create a roadmap that covers the configuration space as good as possible to increase the probability of finding a path, these techniques are generally not good at finding short paths. The reason for this is that, in order to save expensive collision checks, standard PRM does not add cycles to the roadmap graph since these do not contribute to the connectivity of the graph. In this paper we propose a new connection strategy resulting in a graph that has the choice between alternative routes, so that the shortest one can be selected, while still keeping the number of collision checks low. We will prove that the path length converges to a constant times the optimal path length if the sampling time increases. We will also show that the variation in path length of our technique is much lower than that of traditional connection strategies.

Beside its obvious benefit for finding shorter paths, there are a number of other advantages of adding relevant cycles to the graph. It makes the roadmap more robust to dynamic changes in the scene (e.g. the addition of an obstacle). Also, it allows the planner to pick alternative routes which is important to avoid repetitive motion and to help avoid collision when multiple entities move in the same space.

A. Previous work

Most previous techniques use smoothing as a postprocessing step to improve the path. While smoothing is generally good at removing artifacts from the path, it is not good at finding alternative routes that differ considerably from the original one (see Fig. 1 for a trivial example). Kim et al. [17] use an augmented version of Dijkstra's shortest path algorithm to improve all sorts of path optimization problems, but their approach depends on having a well connected graph, which is expensive because of the high number of collision checks. Schmitzberger et al. [21] propose a technique to list all homotopic solutions. But to improve path quality this only works well in 2D since in 3D, solutions are often in the same homotopic class but are so hard to convert into each other that smoothing fails to do so. C-PRM [22] can be used to improve all sorts of variable requirements. Since it postpones most of the collision checks to the query phase, it is very similar to a single shot method. Our approach on the other hand computes alternatives as part of the preprocessing, improving the query time.

B. Paper Structure

In Sections II and III we will explain in more detail the PRM method, and show why it usually fails to find a short path. In Section IV we will present our new connection adding strategy and show that it leads to shorter paths. Next, in Section V we will prove that our technique always converges to the optimal path. Finally, in Section VI we will give empirical results that show that our technique results in shorter paths compared to traditional techniques.

II. THE PRM METHOD

Motion planning is usually performed in configuration space (C space). Every dimension in C space corresponds to a degree of freedom of the robot. All obstacles are transformed into C space. Together the obstacles form the forbidden space

(C_{for}) where the robot is not allowed to move. The space between the obstacles is called the free space (C_{free}). The robot itself is a point in C space. The motion planning problem of a robot among a set of obstacles in workspace is now translated to planning a (collision free) path for a point in C space. A path is collision free if it is entirely contained in C_{free}.

The probabilistic roadmap method tries to create a map that covers C_{free} as good as possible. Over the last years many techniques have been proposed, all based on the same underlying concept.

The main loop of the PRM method selects random configurations c in C_{free}. These configurations are the vertices V in a graph G. For every c, a set of neighbor configurations N_c is selected. Then a local planner is used to try to connect c to each of its neighbors. If the local planner succeeds, the connection is added to the list of edges E in G. The idea is to keep the local planner as simple as possible, allowing for fast collision detection. Typically the local planner checks if the straight line connection between two configurations in C space is collision free. Practically this means that (binary) interpolation between two configurations is used to check if the path is collision free. The construction of the roadmap is shown in pseudo code in Algorithm 1.

Algorithm 1 CONSTRUCTROADMAP

Let: $V \leftarrow \emptyset$; $E \leftarrow \emptyset$;
1: **loop**
2:    $c \leftarrow$ a (random) configuration in C_{free}
3:    $V \leftarrow V \cup \{c\}$
4:    $N_c \leftarrow$ a set of neighbor vertices chosen from V
5:    **for all** $c' \in N_c$ **do**
6:      **if** the line (c, c') is collision free **then**
7:        add the edge (c, c') to E

After adding a number of vertices and edges, the graphs tends to get connected and reflects the connectivity of C_{free}. Now the graph can be used to solve motion planning queries. First the start and goal configurations are added to the graph using the local planner. Then a simple shortest path algorithm (like Dijkstra's algorithm) can be used to find the shortest path between the start and goal configurations.

In order to create the graph, the local planner needs a distance measure between configurations. This is not trivial since the distance does not only depend on the Euclidian distance, but also on the amount of rotation needed to move from one configuration to another. A lot of effort has been put in finding an appropriate distance measure (see for example [2]), but often it is problem dependent.

Since the collision checks are by far the most expensive step in the algorithm, their number has to be kept as low as possible. Adding an edge to a configuration already in the same connected component does not contribute to the exploration of C_{free}, so usually only edges that connect two different connected components are allowed in G.

Many improvements that have been proposed aim at lowering the number of collision checks, and thus lowering the creation time of the graph. Usually this is done by adapting the sampling strategy by allowing only those configurations that contribute to exploring the C_{free} space. Examples of such improvements include [7], [8], [13], [19], [24], [12], [6], [14]. An overview and comparison of numerous sampling and neighbor selecting strategies can be found in [11].

III. CONNECTION STRATEGIES

While many techniques are successful in covering the C_{free} space, they are often not very good at finding short and efficient paths. This is why the query path is usually smoothed as a postprocessing step. In its most basic form, smoothing consists of repeatedly selecting two random configurations c and c' on the query path and sending them to the local planner. If the local planner succeeds, the connection between c and c' is added to the path, and the previous path between c and c' is removed. This is repeated for some time.

Definition III.1 (Homotopy). *Two paths P^0 and P^1 are said to be in the same* homotopic *class only if P^0 can be continuously distorted in P^1 in C_{free}.*

Definition III.2 (Convertibility). *If two paths are in the same homotopic class, and one can be distorted into the other by a series of smoothing steps as described above, the path is said to be convertible to the other.*

By definition, smoothing is good at finding paths that are convertible from the current path (like removing artifacts), but usually cannot find other paths (see Fig. 1 for an example). Since the path resulting from a query is often not convertible to the optimal (shortest) path, traditional techniques don't find a path close to the optimal path.

The reason that a path, found after a query, is often not convertible to the optimal path is that the graph is a forest. This implies that there is at most one path in G from one configuration to another. Thus, the path found after the query phase depends heavily on the random choices that were made in the preprocessing stage.

A number of different connection strategies have been proposed over the last couple of years which can be used to increase the number of alternative paths. The easiest solution would seem to connect every configuration to all of its neighbors. This is called the *nearest-n* method. It simply tries to connect to all nearest n neighbors. While this method finds many alternative routes and thus often succeeds in finding a short path, most connections are redundant and the running time of the algorithm increases.

Component-n tries to connect to n configurations in each connected component in the neighbor set. Many connections do not contribute to exploring the C space and the running time increases because of the many extra collision checks.

IV. ADDING CYCLES TO THE GRAPH

To increase the probability of finding a short path, we need to have alternative routes in the graph. Since adding too many
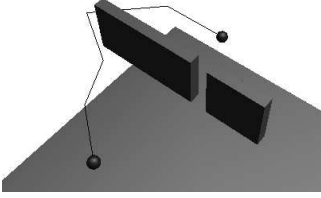
Fig. 1. Even though the shortest path is in the same homotopy class as the current one, they are impossible to convert into each other using standard smoothing.

edges to G would result in a high running time, we would like to add only those edges that have a low probability of being found after smoothing i.e. we add only those edges that have a high probability of adding a path that is not convertible to an existing one. We call such an edge useful.

Definition IV.1 (Useful edge). *Let c be a random chosen configuration in C_{free} and N_c its set of neighbors. c' is a configuration in N_c and $d(c, c')$ is the distance between c and c'. The graph distance between c and c' is $G(c, c')$, this is the length of the shortest path in the graph from c to c'. If there is no path from c to c', $G(c, c')$ is ∞. The edge $E(c, c')$ is K-useful if:*

$$K \cdot d(c, c') < G(c, c') \qquad (1)$$

We will talk about useful edges, where the dependence on K is implicitly understood. So only cycles that improve the graph distance between c and c' by a substantial factor K are added to the graph. Changing the value of K influences the number of cycles that are added to the graph. A small value of K adds more cycles, a large value of K adds less. If K is smaller than 1, all edges are allowed, since $G(c, c')$ can never be smaller than $d(c, c')$. If $K = \infty$ no cycles are allowed and the resulting graph is a forest.

A major algorithmic question is how to compute $G(c, c')$. Since adding a new edge to the graph can influence all existing graph distances, these cannot be maintained and a shortest path algorithm (like Dijkstra) has to be executed for every potential connection, in order to calculate $G(c, c')$. When the number of vertices in the graph becomes large, the calculation of Dijkstra will become a dominating factor in the running time.

To speed this up we proceed in a way similar to the A^* algorithm. Suppose we want to know if edge $E(c, c')$ is a useful edge. Dijkstra calculates the shortest graph distance from the source to all other vertices. But we don't need the exact distance; we are only interested in whether $G(c, c')$ is larger than $K \cdot d(c, c')$. Since Dijkstra visits vertices in increasing graph distance from c, we know that when vertex v is selected by the algorithm (and c' has not yet been selected), $G(c, c')$ is at least $G(c, v)$. We can use this property to prune the search, let:

$$G_d(c, v) = G(c, v) + d(v, c') \qquad (2)$$

We now let Dijkstra select vertices based on the value of G_d. If a vertex v is selected, it gets a value equal to the shortest

graph distance from that configuration to c. $G_d(c, v)$ is a lower bound of $G(c, c')$, because, once Dijkstra selects a vertex v, we know that it takes at least another $d(v, c')$ to reach c'.

If $G_d(c, v)$ of a vertex v is larger than $K \cdot d(c, c')$ we can stop the calculation of Dijkstra, because we know that $G(c, c')$ will be at least $G_d(c, v)$. In this case the edge (c, c') will be added to the graph. If we reach c' before the threshold is reached, we also stop the search and do not add the edge.

The resulting graph is no longer a tree because of the cycles. We will call this prunded version of Dijkstra the *usefulness test*. In the query phase the normal version of Dijkstra's algorithm can be used to find the shortest path in the graph that is hopefully convertible to the actual shortest path. Afterward, standard smoothing can be used to optimize the path.

In practice there is a maximum number of neighbors (M_n) to which a configuration is connected. If V is the collection of vertices, then, after preprocessing, the maximum number of edges in the graph, is $M_n \cdot |V|$. During preprocessing the worst case running time of Dijkstra increases linear with the number of vertices in the graph.

An alternative to the usefulness test is using a dynamic all pairs shortest path algorithm [9], [10]. Because our experiments (see Section VI) show that the usefulness test is not the dominating factor in all but the simplest scenes, we did not implement this.

V. THEORETICAL RESULTS

In this section we will show that, using the usefulness property, the path length of our technique converges to K times the optimal path length.

Let $cl > 0$ be a small clearance and let Π^{cl} be the optimal path with clearance cl. We assume that cl has been chosen small enough such that Π^{cl} exists. Let $len(\Pi^{cl})$ denote the length of this path. We will show that, when time goes to infinity, our approach will find a path with a length converging to $K \cdot len(\Pi^{cl})$.

Let $\delta > 0$ be a sufficiently small constant. We will approximate the path Π^{cl} with (hyper-) cylinders of radius δ. Choose the cylinder length l such that

$$\sqrt{l^2 + \delta^2} < cl \qquad (3)$$

We pick points Π_i on Π^{cl} as follows: Π_0 is the start point of the path. Next, as Π_1 we pick the first point on the path that has a straight-line distance l from Π_0. Π_2 is the first point at distance l from Π_1, etc. (See Fig. 2(a).) Finally we add the goal point of the path. Assume in total we added $n+1$ points. We approximate Π^{cl} with n cylinders. The centers of their bottom and top are the points Π_i (Fig. 2(b)) and the cylinders have radius δ.

Lemma V.1 (Collision free cylinder). *The cylinders as defined above are guaranteed to be collision free.*

Proof: Let Π_i be the start point of the cylinder. Let d be the distance from Π_i to the furthest point in the cylinder. By the choice of l and δ we have

(a) The path Π^{cl} divided in parts using points Π_i.

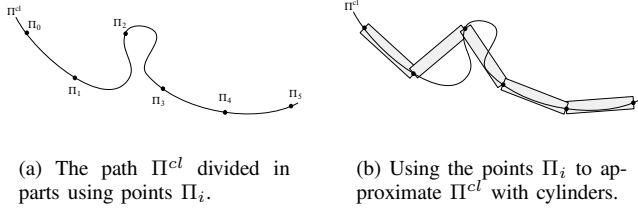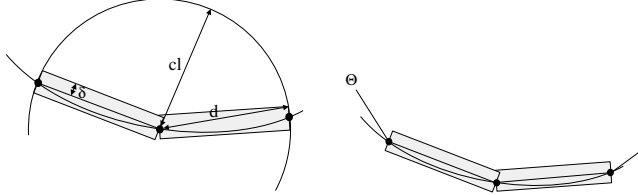(b) Using the points Π_i to approximate Π^{cl} with cylinders.

Fig. 2.

(a) Because d is smaller or equal than cl, the cylinders are collision free.

(b) Connecting the points Π_i to create the path Θ.

Fig. 3.

$$d = \sqrt{l^2 + \delta^2} < cl \qquad (4)$$

As point Π_i lies on the path Π^{cl} it has a clearance of at least cl. Hence the cylinder must be collision free. (See also Fig. 3(a)). ∎

Consider the path Θ that is created by connecting every Π_i to Π_{i+1} by a straight line motion in configuration space (Fig. 3(b)). It is easy to see that

$$len(\Theta) \leq len(\Pi^{cl}) \qquad (5)$$

Now assume that after long enough sampling, each cylinder i contains a sample c_i. Let Θ_i be the closest point on Θ from c_i (Fig. 4(a)). Consider path Φ that is constructed by connecting every Θ_i to Θ_{i+1} with a straight-line motion (Fig. 4(b)). Since Θ_i and Θ_{i+1} are both within the collision-free sphere of Fig. 3(a), the straight line path (Θ_i, Θ_{i+1}) is also collision free. It is easy to see that

$$len(\Phi) \leq len(\Theta) \qquad (6)$$

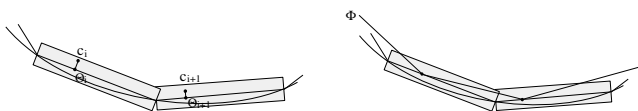Let c_i and c_{i+1} be the samples in two consecutive cylinders. Clearly their distance can be bounded as follows:

(a) The point Θ_i is the closest point on Θ from c_i.

(b) Connecting the points Θ_i to create the path Φ.

Fig. 4.
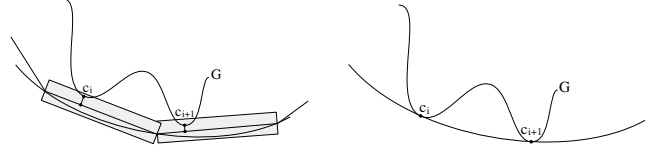
(a) The path G.

(b) If the radius δ of the cylinders approaches 0, the points c_i approach Π^{cl}.

Fig. 5.

$$d(c_i, c_{i+1}) \leq 2 \cdot \delta + d(\Theta_i, \Theta_{i+1}) \qquad (7)$$

Since the straight-line connection between c_i and c_{i+1} is collision-free, we know from the usefulness property that there is a bound on their graph distance (Fig. 5(a)):

$$G(c_i, c_{i+1}) \leq K \cdot d(c_i, c_{i+1}) \leq K \cdot (2 \cdot \delta + d(\Theta_i, \Theta_{i+1})) \quad (8)$$

Because Equation 8 holds for every segment we can bound the graph length as follows:

Theorem V.1 (Graph length). *The graph distance $G(s, g)$ between two arbitrary graph configurations s and g consisting of n edges is bounded by:*

$$K \cdot (2 \cdot n \cdot \delta + len(\Phi)) \leq K \cdot \left(2 \cdot n \cdot \delta + len(\Pi^{cl})\right) \quad (9)$$

If we decrease the radius of the cylinders and thus let δ approach to 0 (Fig. 5(b)), the length of the graph path between c_0 and c_n approaches $K \cdot len(\Pi^{cl})$. When cl approaches 0 as well, the length approaches K times the length of the optimal path Π^0.

For each $cl > 0$ and $\delta > 0$ the cylinders have an equal constant-size area. So if time goes to infinity the chance that each cylinder indeed contains a sample approaches 1. (Actually, this is not true for the last cylinder but that cylinder already contains the goal configuration as a sample).

VI. Experimental Results

Our algorithm has been implemented in (Visual) C++ using our motion planning system SAMPLE (System for Advanced Motion PLanning Experiments). All tests were run on a Pentium 2.4GHz with 1GB internal memory. We used SOLID 3.5 [5] for collision detection. To test our algorithm we used several benchmark scenes (see Figs. 6 to 9). All tests consisted of a preprocessing and a postprocessing (smoothing) phase.

Scene 1 This is a scene with a number of long boxes that are slightly shorter than the scene height. The robot used is a cylinder that needs rotation to get through the passages between the boxes. Since the right choice for a passage in every row of obstacles is crucial for a short path, we expect our algorithm to outperform standard PRM.

Scene 2 This scene consists of 500 randomly distributed tetrahedra. The robot is an L-shaped object which needs a lot of rotation to maneuver through the scene. The whole scene consists of one convertible class and we expect smoothing to

work well in this scene, so we don't expect a big difference between the two methods.

Scene 3 The robot is a complicated flamingo figure consisting of 7049 polygons. It has to maneuver out of a cage (1032 triangles). Since the choice from which hole to leave the cage is crucial, we expect our algorithm to outperform standard PRM.

Scene 4 Here, a simple sphere object has to move from one position to another in a complicated house scene consisting of 1600 polygons. There are roughly two solutions for this scene; one that uses the interior of the house, and a longer one that uses the "garden." Since smoothing will probably not be able to convert one type of path into the other, the choice for the right type of path is important.

First we need to establish the best K value for our algorithm. Connecting every configuration to all of its neighbors (equal to a K value of 1) results in the best graph and thus the shortest paths. But the price to pay for the increased running time is huge. Once the standard PRM finds a path, further preprocessing time will not help in improving the path length. For the cycle method additional preprocessing time will lead to additional, potentially shorter paths. This means that for a fair comparison the running time is crucial. On the other hand it depends on the application of the motion planning problem which price you are willing to pay for a shorter path.

In our experiments we resolved this issue as follows. For every scene we established the number of vertices (n) for which standard PRM always ($> 99\%$) finds a path. Then, we used this value to find an optimal K value. We let K increase from 1.0 to 30.0 using a step size of 1 for $K \leq 10$ and a stepsize of 4 for $K > 10$ and measured the running time and query path length after creating n configurations. All other parameters were fixed. From previous experiments we know that for scene 1, connecting to a maximum of 10 neighbors is optimal. For scenes 2 and 4 we used a value of 15 and for scene 3 we used a maximum of 20. The maximum neighbor distance was half the size of the bounding box in every scene. The local planner used binary interpolation to check an edge for collision. The distance between two configurations was calculated by taking the radius of the robot, multiplying this with the rotation angle and adding the Euclidian distance.

As an example, we show the graph for scene 3 (Fig. 10). As can be seen from the graph, the lower the value of K, the more edges are being added to the scene and thus the more time the collision checks take. This results in a higher total running time and smaller query distance. For example for a K value of 6, the preprocessing time is about 7.5 seconds, the average path length is 200 and the smoothed path length is 100. There is no such thing as *the* optimal K value, it depends on the price the user is willing to pay for a shorter path. Here, we choose a K value of 6 for scene 3.

We also ran this test for several other scenes. They all have their own optimal K value. Which K value this is, is dependent on the complexity of the scene and the preferences of the user. For every potential connection a decision has to
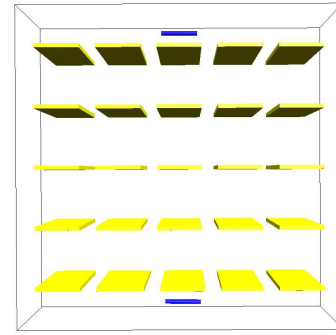

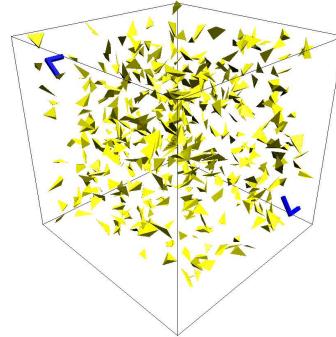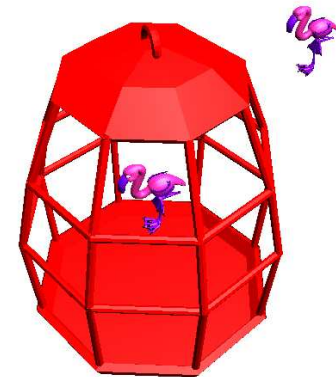Fig. 6. Test scene 1.
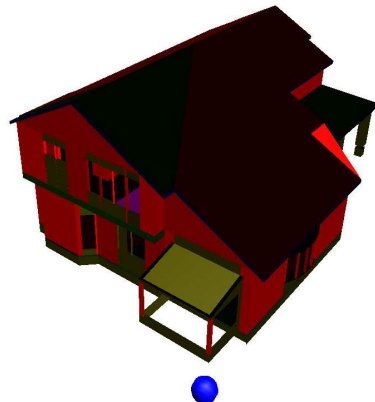

Fig. 7. Test scene 2.

Fig. 8. Test scene 3.

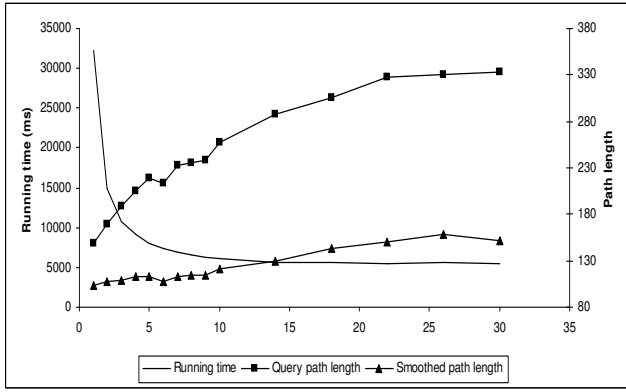Fig. 9. Test scene 4 (second query point inside the house).

Fig. 10. Establishing an optimal K value for scene 3. For every point in the graph we determined the running time, the query path length and the smoothed path length. Every value is the average of 100 runs.

be made whether to add it or not. If both configurations are in the same connected component, the usefulness test has to be initialized and executed to make the decision. If the complexity of the scene is very low, the usefulness test costs a lot of time compared to the collision checks. In these circumstances it is better to prune the usefulness test quickly (by using a low K value) and add the edge if it is collision free. If the scene is more complex, collision checks are more expensive and dominate the running time. These scenes have a higher optimal K value. Although, for a fair comparison, in our experiments we calculated an exact value for K, we had no problem estimating a good K value since the range of K values for which the algorithm performs well is broad (see Fig. 10). After preprocessing, the query paths were smoothed using standard smoothing. The smoothing time was chosen such that more smoothing did not yield further improvement of the path length.

Scene 1 We created a graph of 180 configurations. Experiments showed that using a postprocessing time (smoothing) of more 0.2 seconds, did not result in shorter paths, so we used a value of 0.2 seconds. We chose a K value of 3 here. The results for this scene are shown in Fig. 11(a) and 11(b). In the graphs we show the path length relative to *the* shortest path length. The lines in the graphs show the minimum and maximum values. The boxes show the area between the first and third quartiles. The square shows the average value. As can be seen from the results, our technique performs better in this scene compared to standard PRM. Also the variation in path length is much smaller by allowing useful cycles. But when looking at the results after smoothing we conclude that the results are not as good as we expected beforehand. The reason for this is that, in this scene, smoothing is sometimes able to convert between paths that take a different route between the boxes. Also, collision checks are relatively cheap in this scene. Thus, the usefulness test is relatively expensive and the running time of the cycle method is higher (800ms vs 450ms on average).
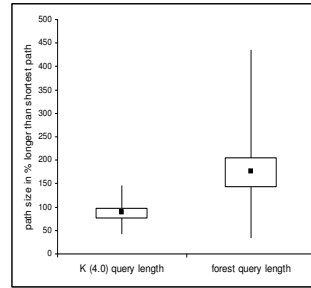
Scene 2 Here we added 250 configurations to the graph. After each run, the path was smoothed for 1 second. The results are shown in Fig. 11(c) and 11(d). Again it is clear that adding useful cycles lowers both the path length and
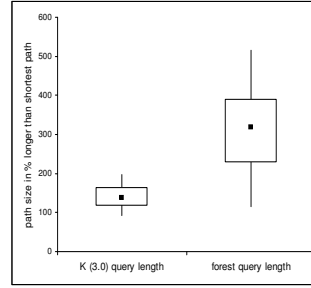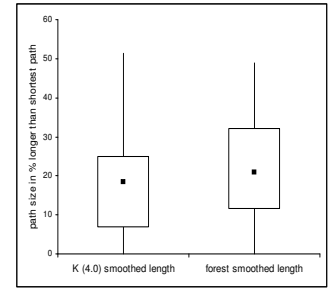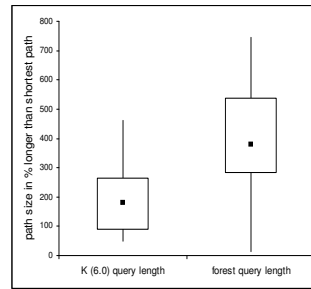

(a) Scene 1: query path.
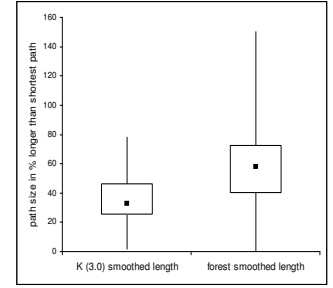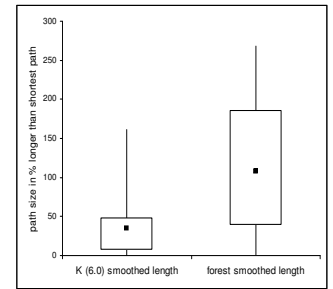

(b) Scene 1: smoothed path.
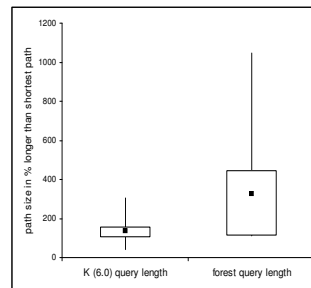

(c) Scene 2: query path.
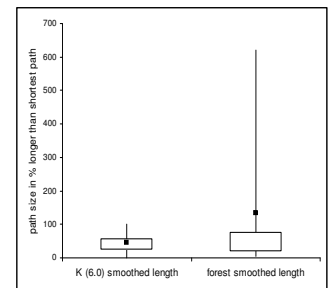

(d) Scene 2: smoothed path.

(e) Scene 3: query path.

(f) Scene 3: smoothed path.

(g) Scene 4: query path.

(h) Scene 4: smoothed path.

Fig. 11. The results of the various scenes. The boxes show the area between the first and third quartiles. The lines show the highest and lowest value. The black squares show the average values.

the variation in path length. In this scene it is very easy for smoothing to find shorter paths. This is why the average path length after smoothing for both techniques does not differ a lot. Still, the variation in the smoothed path length is slightly lower when cycles are used. The average running time of the cycle method was about 1 second higher (3.3s vs 2.3s).

Scene 3 We added 150 configurations to the graph, followed by 2 seconds of standard smoothing. The key problem in this scene is finding a path through the correct hole in the cage. If this hole is found, then standard smoothing is able to remove the rough edges and redundant rotation in the path. Since a specific rotation is needed to get through a hole, smoothing will usually keep the path in the same homotopy class. There is a huge improvement in path length variation, even after smoothing. See Fig. 11(e) and 11(f) for the results. The running time of the cycle method was 7.4 seconds on average; that of the forest method was 5.5 seconds on average.

Scene 4 A graph was created consisting of 350 configurations, followed by a smoothing phase of 1 second. Roughly speaking there are two solutions for this scene: a short one that stays inside the house, and a long one that uses the "garden." Once the forest method has found a path that uses the garden, it will never find the short one through the house. The cycle method is able to improve its path, by introducing a cycle and thus finding both paths. This is also the main reason that the average path length of the forest method is large; about one out every five paths uses the "garden." Smoothing will never be able to convert between the two types of paths, so even after smoothing the average path length of the forest method is much longer. Again the variation in path length is smaller when using useful cycles. The results for scene 4 are shown in Fig. 11(g) and 11(h). The average running time of the cycle method was slightly larger (11.0s vs 9.5s) because collision checks are time-consuming in this scene.

VII. CONCLUSIONS

We presented a new connection technique for probabilistic roadmaps that decreases the average path length and also decreases the variance of the path length without increasing the preprocessing time considerably. It works well in scenes where standard smoothing is unable to convert a long path into a short one. In very simple scenes the difference is not so large because the running time of the repeatedly executed usefulness test tends to dominate the overall running time. In more complex scenes the technique works much better and both the path length and its variation decrease a lot. Having alternative routes is also important for other applications. For example it allows for variation in the routes computer entities in a game take. Also it can be used to avoid deadlock situations when multiple robots move in the same environment. Finally, it is useful in dynamic scenes where additional obstacles might appear. We are currently investigating these applications.

ACKNOWLEDGMENT

REFERENCES

[1] N. Amato, O. Bayazit, L. Dale, C. Jones, D. Vallejo, OBPRM: An obstacle-based PRM for 3D workspaces, in: P.K. Agarwal, L.E. Kavraki, M.T. Mason (eds.), *Robotics: The algorithmic perspective*, A.K. Peters, Natick, 1998, pp. 155–168.

[2] N. Amato, O. Bayazit, L. Dale, C. Jones, D. Vallejo, Choosing good distance metrics and local planners for probabilistic roadmap methods, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1998, pp. 630–637.

[3] N. Amato, Y. Wu, A randomized roadmap method for path and manipulation planning, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1996, pp. 113–120.

[4] J. Barraquand, L. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, P. Raghavan, A random sampling scheme for path planning, *Int. Journal of Robotics Research* **16** (1997), pp. 759–774.

[5] G. van den Bergen, Collision Detection in Interactive 3D Environments, *Morgan Kaufmann* In press 2003.

[6] M. Branicky and S. Lavalle and K. Olson and L. Yang, Quasi randomized path planning, *IEEE Int. Conf. on Robotics and Automation,* 2001.

[7] V. Boor, M.H. Overmars, A.F. van der Stappen, The Gaussian sampling strategy for probabilistic roadmap planners, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1999, pp. 1018–1023.

[8] R. Bohlin, L.E. Kavraki, Path planning using lazy PRM, *Proc. IEEE Int. Conf. on Robotics and Automation,* 2000, pp. 521–528.

[9] C. Demetrescu, G.F. Italiano, Fully Dynamic All Pairs Shortest Paths with Real Edge Weights, *IEEE Symp. on Foundations of Computer Science,* 2001, pp. 260-267.

[10] C. Demetrescu, G.F. Italiano, A New Approach to Dynamic All Pairs Shortest Paths, *Proc. 35th Annual ACM Symposium on Theory of Computing* 2003.

[11] Roland Geraerts, Mark H. Overmars, A Comparative Study of Probabilistic Roadmap Planners, *Proceedings WAFR 2002-2003* 2002, pp. 40–54.

[12] David Hsu and Tingting Jiang and John Reif and Zheng Sun, The Bridge Test for Sampling Narrow Passages with Probabilistic Roadmap Planners, *Proc. IEEE Int. Conf. on Robotics and Automation,* 2003.

[13] D. Hsu, L. Kavraki, J.C. Latombe, R. Motwani, S. Sorkin, On finding narrow passages with probabilistic roadmap planners, in: P.K. Agarwal, L.E. Kavraki, M.T. Mason (eds.), *Robotics: The algorithmic perspective*, A.K. Peters, Natick, 1998, pp. 141–154.

[14] P. Isto, Constructing Probabilistic Roadmaps with Powerful Local Planning and Path Optimization, *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems,* 2002, pp. 2323–2328.

[15] L. Kavraki, Random networks in configuration space for fast path planning, *PhD thesis,* Stanford University, 1995.

[16] L. Kavraki, J.C. Latombe, Randomized preprocessing of configuration space for fast path planning, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1994, pp. 2138–2145.

[17] J. Kim, R. Pearce, N. Amato, Extracting optimal paths from roadmaps for motion planning, unpublished.

[18] J. C. Latombe, Robot motion planning, *Kluwer Academic Publishers, Boston,* 1991.

[19] C. Nissoux, T. Siméon, J.-P. Laumond, Visibility based probabilistic roadmaps, *Proc. IEEE Int. Conf. on Intelligent Robots and Systems,* 1999, pp. 1316–1321.

[20] M.H. Overmars, A random approach to motion planning, *Technical Report RUU-CS-92-32, Dept. Comput. Sci., Utrecht Univ.,* Utrecht, the Netherlands, 1992.

[21] E. Schmitzberger, J.-L. Bouchet, M. Dufaut, W. Didier, R. Husson, Capture of homotopy classes with probabilistic road map, *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems,* 2002.

[22] G. Song, Shawna Miller, N. M. Amato, Customizing PRM Roadmaps at Query time, *Proc. IEEE Int. Conf. on Robot. Autom. (ICRA)* 1999, pp. 2958–2963.

[23] P. Švestka, Robot motion planning using probabilistic roadmaps, *PhD thesis, Utrecht Univ.,* 1997.

[24] S.A. Wilmarth, N.M. Amato, P.F. Stiller, MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1999, pp. 1024–1031.