Path Planning in Practice; Lazy Evaluation on a Multi-resolution Grid

Robert Bohlin
Department of Mathematics
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Abstract

We present a resolution complete path planner based on an implicit grid in the configuration space. The planner can be described as a two-level process in which a global planner restricts a local planner to certain subsets of the grid. The global planner starts by letting the local planner search in a coarse subset of the grid, and successively refines the grid until a solution is found. The local planner applies a scheme for lazy evaluation on each subgrid in order to minimize collision checking and thereby increase performance.

Experimental results in an industrial application show that lazy evaluation on a grid is very efficient in practice. The algorithm is particularly useful in high dimensional, relatively uncluttered configuration spaces, especially when collision checking is computationally expensive. Single queries are handled quickly since no preprocessing is required.

1 Introduction

Path planning for robots has received much attention over the last decades. The general problem is to find collision-free paths for a robot in an environment containing obstacles. Algorithms for its solution are, however, rarely used in practice due to their computational complexity.

To be useful in practice the planner must be fast, in particular for easy problems, and generate paths that are short and smooth enough to be executed by a real robot. In industrial applications, the geometry of the robot and its environment is typically very complex, making collision checking computationally expensive. Since many existing planners rely on fast collision checking, their practical use in these situations is limited.

Our aim with this paper is to meet the requirements above and design a planner that is useful in industrial applications. By using an implicit multi-resolution grid combined with a lazy evaluation technique, we can reduce collision checking and thereby increase speed.

Before describing the algorithm, we introduce some notation and give a brief overview of existing planners.

1.1 Notation

We let $\mathcal{W}$ denote a subset of $\mathbf{R}^2$ or $\mathbf{R}^3$ in which a robot $\mathcal{A}$ is moving. The position of $\mathcal{A}$ is described by a *configuration* $\boldsymbol{q}$ such that the position of every point on $\mathcal{A}$ can be determined relative to a fixed frame in $\mathcal{W}$. The set of all configurations is called the *configuration space* and is denoted by $\mathcal{C}$. For a configuration $\boldsymbol{q} \in \mathcal{C}$, $\mathcal{A}(\boldsymbol{q})$ denotes the subset of $\mathcal{W}$ occupied by $\mathcal{A}$.

The cardinality of $\mathcal{C}$ is generally infinite since the robot is assumed to move continuously in $\mathcal{W}$. In what follows we assume that $\mathcal{C}$ can be identified with a subset of $\mathbf{R}^d$, where d is equal to the number of degrees of freedom (dof) of $\mathcal{A}$. For convenience we let $\mathcal{C}$ also denote this subset of $\mathbf{R}^d$, thus $\boldsymbol{q}$ also denotes a point in $\mathbf{R}^d$. For example if $\mathcal{A}$ is an articulated robot arm, we can let $\mathcal{C} = I_1 \times \cdots \times I_d$, where I_j is the range of joint j.

The aim of path planning is to avoid a set of obstacles $\mathcal{O}$ in $\mathcal{W}$. If $\mathcal{A}$ intersects $\mathcal{O}$ we say that $\mathcal{A}$ *collides* with the obstacles, and we define the mapping $\Phi : \mathcal{C} \to \{0, 1\}$ as

$$\Phi(\boldsymbol{q}) = \begin{cases} 0 & \text{if} \quad \mathcal{A}(\boldsymbol{q}) \bigcap \mathcal{O} \neq \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

This mapping, which is called the *collision checker*, divides the configuration space into two disjoint sets $\mathcal{E}$ and $\mathcal{F}$ such that $\mathcal{E} = \Phi^{-1}(0)$ is the set of colliding configurations and $\mathcal{F} = \Phi^{-1}(1)$ is the set of collision-free (or *feasible*) configurations.

Given an initial configuration $\boldsymbol{q}_{init} \in \mathcal{F}$ and a final configuration $\boldsymbol{q}_{goal} \in \mathcal{F}$, we define the path planning problem as follows: Find a continuous path $\gamma : [0, 1] \to \mathcal{C}$ such that $\gamma(0) = \boldsymbol{q}_{init}$, $\gamma(1) = \boldsymbol{q}_{goal}$ and $\gamma(t) \in \mathcal{F}$ for all $t \in [0, 1]$, or determine that no such path exists. An algorithm that solves this problem, or a variation of it, is called a *path planner* or simply a *planner*.

To measure distances in $\mathcal{C}$ we need a metric ρ_{path}. This metric is also used to measure lengths of paths. For simplicity paths are ranked by length, and we prefer short paths with respect to ρ_{path}. So, by defining this metric, we decide which paths are of high quality and which paths are of poor quality.

1.2 Previous work

The path planning problem has been extensively studied in the last decades, and a number of different approaches are proposed; see [7, 11, 15] for overviews. An algorithm is called *complete* if it always will find a solution or determine that none exists. However, due to the complexity of the path planning problem, complete planners are too slow to be useful in practice [5].

Another category of planners discretize the configuration space. If these planners are complete in the limit as the discretization approaches a continuum, they are called *resolution complete*. See [6, 8, 14] for resolution complete planners.

A general problem of discretizing the configuration space is that the memory requirement grows rapidly with the dimension. In [8], this problem is solved by an *implicit* representation of a grid. A version of the A*-algorithm [17] is then used to find a feasible path in the grid. They also show different ways of choosing the discretization of the configuration space.

Trading completeness for speed, randomized techniques have been successfully applied to many problems in high-dimensional configuration spaces. The Randomized Path Planner (RPP) in [2] uses a potential field as guidance towards the goal, and random walks to escape local minima.

The Ariadne's clew algorithm in [18] incrementally builds a tree of feasible paths using genetic optimization. Considering the initial configuration as a landmark, the planner finds a path from one of the landmarks to a point as far as possible from all previous landmarks. A new landmark is placed at this point. New landmarks are placed until the goal configuration can be connected to the tree.

The Probabilistic Roadmap Method (PRM) [12, 13, 20] is a method that has been shown to work well in practice in high-dimensional configuration spaces. The idea is to represent and capture the connectivity of $\mathcal{F}$ by a random network, a *roadmap*, whose nodes correspond to randomly selected configurations. If a local planner finds a feasible path between two nodes, they are connected by an edge. See also [1, 9] for methods to increase the connectivity of the roadmap. If the start and goal configurations can be connected to the same component of the roadmap, then a solution has been found. PRM is particularly useful for multiple queries, since once an adequate roadmap has been created, queries can be answered very quickly.

Lazy PRM in [3] is a probabilistic roadmap planner well suited for single queries. The underlying idea is to minimize collision checking by introducing a scheme for lazy evaluation of the nodes and edges in the roadmap. The scheme is particularly useful when collision checking is expensive, for example in industrial applications with complex geometry. See also [19] for a related technique. A recent approach using quasi-random sampling is described in [4]

Other methods, described in [10] and [16], build two trees rooted at the initial and goal configurations respectively. In [10], the trees are expanded by generating new nodes randomly in the vicinity of the two trees, and connecting them to the trees by a local planner. The planner in [16] iteratively generates a configuration, an attractor, uniformly at random in $\mathcal{C}$. Then, for both trees, the node closest to the attractor is selected and a local planner searches for a path of a certain maximum length towards the attractor. A new node is placed at the end of both paths. The process stops when the two trees intersect.

1.3 Information collection

A path planner may obtain information about the configuration space, or rather the obstacles in the configuration space, in different ways. Canny [5] represents the boundary of $\mathcal{F}$ by a set of algebraic equations giving complete information about the obstacles in $\mathcal{C}$. Unfortunately, they are very complex, and are difficult to use in practice.

A simpler, but much less informative way, is to sample at certain points in $\mathcal{C}$. To evaluate Φ at a configuration $\boldsymbol{q}$, we determine the position and orientation of all links of the robot and check whether or not they intersect the obstacles $\mathcal{O}$. The minimum distance δ to the obstacles gives somewhat more information. If $\delta > 0$, then it is possible to determine a ball around $\boldsymbol{q}$ which is entirely in $\mathcal{F}$, and if $\delta \leq 0$, then $\boldsymbol{q} \in \mathcal{E}$.

Most of the resolution complete planners and roadmap planners mentioned in Section 1.2 use either of the two sampling methods above. For simple robots and obstacles they are relatively straightforward, but for robots and obstacles with complex geometric descriptions (e.g. thousands of polyhedra) they are computationally expensive. In particular the latter method, to find the minimum distance to the obstacles, become very complex when the geometric model contains curved surfaces.

The planner we describe in this paper can be used with either of the above sampling methods. Our aim, however, is to keep the planner as simple and general as possible. Therefore we only calculate intersections in $\mathcal{W}$ and not distances, i.e., we require that the planner only obtains information by evaluating Φ point-wise.

2 The Algorithm

The path planner presented in this paper is based on a discretization of the configuration space. A dense graph (a grid) $\mathcal{G}$ is placed in $\mathcal{C}$, and only paths contained in the graph are considered. We assume that $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ have been specified, and for simplicity we assume that they coincide with two nodes (also denoted $\boldsymbol{q}_{init}$

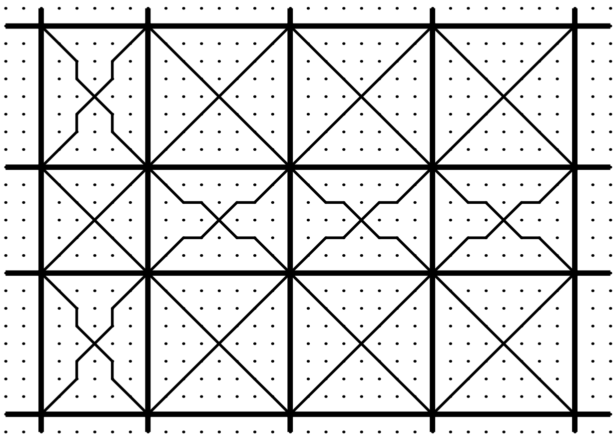


Figure 1: Example of a 35×25 grid defining $\mathcal{G}$ in a two-dimensional configuration space. Thick lines show enabled hyper planes and thin lines show diagonal edges of $\mathcal{G}'$. Edges of $\mathcal{G}$ are omitted.

and $\boldsymbol{q}_{goal}$) in $\mathcal{G}$. Unless otherwise stated, a *path* will always refer to a path in $\mathcal{G}$ connecting $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$.

The planner is resolution complete in the sense that if a feasible path exists in the graph, it will be found. The essential is to find a feasible path in a minimum number of evaluations of Φ.

2.1 Algorithm overview

The approach to search in $\mathcal{G}$ for a feasible path can, as many other planners (e.g. [6, 13, 18]), be described as a two-level planning process. A global planner on the top level restricts the local planner on the lower level to a certain subset $\mathcal{G}'$ of $\mathcal{G}$. The subset $\mathcal{G}'$ is successively extended until a solution is found or $\mathcal{G}' = \mathcal{G}$. If the local planner still fails in the latter case, then no solution exists in $\mathcal{G}$ and the planner returns failure.

The motivation for using subsets of $\mathcal{G}$ is to let the local planner search for simple solutions first. A relatively coarse subset of $\mathcal{G}$ is often sufficient to find a solution in many practical situations, so the scheme of refining $\mathcal{G}'$ makes the planner efficient for simple as well as more difficult planning tasks. Before going into the details of the global and local planners, we need to describe the representation of $\mathcal{G}$ and $\mathcal{G}'$.

2.2 Configuration space representation

To simplify the presentation, we assume that $\mathcal{C}$ is an axis aligned rectangle in $\mathbf{R}^d$. The nodes in $\mathcal{G}$ are defined by the points in a rectangular grid of size $n_1 \times \cdots \times n_d$. We would like to traverse $\mathcal{G}$ parallel with the coordinate axes as well as along diagonals, so we add the appropriate edges. That is, each interior node gets $3^d - 1$ neighbors.

The resolution specified by the parameters $\{n_i\}_{i=1}^d$ is the finest resolution that will be used in $\mathcal{C}$, and determines, for example, the step size with which path segments are checked; if adjacent nodes are in $\mathcal{F}$, we consider the edge between them as being feasible. This means that Φ is only evaluated at nodes in $\mathcal{G}$.

The dimension d of $\mathcal{C}$ is often high and the number of grid points grows rapidly with the dimension, so there is no way to explicitly represent $\mathcal{G}$ in a computer. (In our experiments presented in the next section we let n_i be up to 255, giving more than 10^{14} nodes.) A convenient way of implicitly representing the underlying grid is to define each grid point as the intersection of d hyper planes. For each dimension i, place n_i planes equally spaced in $\mathcal{C}$ and perpendicular to the i:th coordinate axis. Then we get $\prod_{i=1}^d n_i$ intersections between d planes which define the grid points.

2.3 Subgraphs

We can get a sparser graph $\mathcal{G}'$ by disabling some of the planes in the underlying grid, and define the nodes in $\mathcal{G}'$ as the intersections of the enabled planes only, see Figure 1. Note that the only planes that *must* be enabled are the planes that contain $\boldsymbol{q}_{init}$ or $\boldsymbol{q}_{goal}$; if these planes are disabled, $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ will not be nodes in $\mathcal{G}'$.

In $\mathcal{G}'$ we also introduce edges so that we can traverse the graph parallel with the coordinate axes and along diagonals. Note that since we introduced a number of new edges, $\mathcal{G}'$ is no longer a subgraph of $\mathcal{G}$, but if we instead associate each edge of $\mathcal{G}'$ with the sequence of nodes in G that is "covered" by the edge, we can consider $\mathcal{G}'$ as a subgraph of $\mathcal{G}$, see Figure 1.

2.4 Evaluated nodes

As the planning process proceeds Φ will be evaluated at more and more nodes, but since $\mathcal{G}$ is implicitly represented we cannot associate unique information to each node. However, only evaluated nodes need to be represented; one set of feasible nodes and one set of colliding nodes. These sets are kept updated at any time in order to avoid any node being evaluated more than once. The colliding nodes can be seen as nodes deleted from $\mathcal{G}$.

Since evaluating Φ is expensive (≈ 0.08 seconds on average in our test example in Section 3), the number of nodes that within reasonable running time can be evaluated is only a small fraction of the total number of nodes in $\mathcal{G}$. Thus, these sets of nodes will never be too large.

2.5 Global Planning

The global planner controls to which subgraph $\mathcal{G}'$ the local planner is restricted. By starting with a sparse subgraph $\mathcal{G}'$ defined by a only a few enabled planes, the local planner quickly solves many easy planning tasks. If no solution exists in $\mathcal{G}'$, the global planner refines the grid by enabling a few more planes. If all planes are enabled and the local planner still cannot find a solution, then there is no feasible path in $\mathcal{G}$.

In our experiments presented in Section 3, we apply the planner to an articulated robot arm with six dof. The following simple strategy of enabling hyper planes show great performance. Initially, we enable the planes containing $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, and in each of the dimensions 1,2 and 3, we enable another six planes as evenly as possible. (This generally gives $(2+6)^3 \cdot 2^3 = 4096$ nodes in $\mathcal{G}'$.) Each time the local planner fails, we enable the plane in $\mathcal{G}$ that is furthest from an enabled plane. Here, of course, the distances are measured between parallel planes only.

2.6 Local Planning

The local planner is given a subgraph $\mathcal{G}' \subset \mathcal{G}$ from the global planner and starts an exhaustive search for a solution. The technique is similar to the scheme of lazy evaluation presented in [3], i.e., the aim is to find the shortest (with respect to the metric ρ_{path}) feasible path in $\mathcal{G}'$ in a minimum number of evaluations of Φ.

Based on the current status of $\mathcal{G}'$, in which some nodes have been evaluated and others have not, the local planner picks the shortest path $\mathcal{P}$, called a candidate path. Recall that all nodes that have been evaluated to collision are deleted from $\mathcal{G}$. The candidate path is then checked for collision according to a certain scheme described below. As soon as a collision is detected, we know that this is not the path we are looking for, so we delete the colliding node and pick a new candidate path. This procedure is repeated until a feasible path is found or no candidate path exists in $\mathcal{G}'$.

When checking the path, we first evaluate the nodes. Starting respectively with the first and the last node on $\mathcal{P}$ and working toward the center, we alternately evaluate the nodes along the path. If all nodes are feasible then we check the edges on $\mathcal{P}$, first with a coarse resolution and then we do stepwise refinements until all edges are feasible. As soon as a collision is found, we delete the corresponding node from $\mathcal{G}$ (recall that edges of $\mathcal{G}'$ refers to a sequence of nodes of $\mathcal{G}$, see Figure 1), and reject $\mathcal{P}$.

The hope when applying this scheme to a candidate path is of course that it is feasible. On the other hand, if it is not feasible, we would like to detect that as quickly as possible in order to avoid evaluating nodes on a path that will be rejected anyway. At the same time we want to reject as many *other* colliding candidate paths as possible. The following two observations motivate the procedure of evaluating nodes before edges, and checking nodes from outside in. First, a colliding node in $\mathcal{G}'$ will cut away more than a colliding edge in $\mathcal{G}'$. Second, a node close to $\boldsymbol{q}_{init}$ or $\boldsymbol{q}_{goal}$ is likely to cut away a larger portion of the colliding candidate paths than a node far away from $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$.

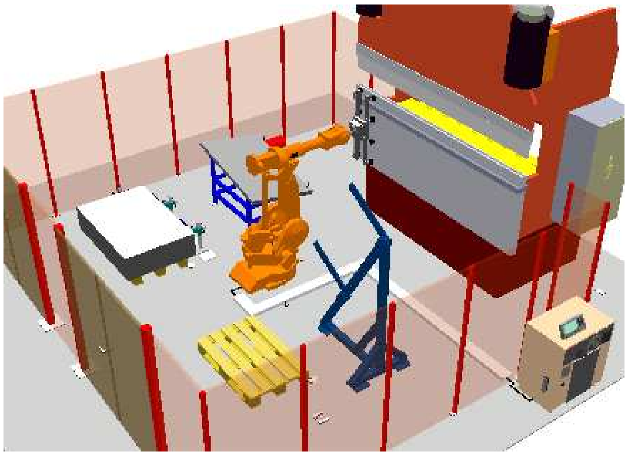


Figure 2: The workcell used in our experiments. The robot is in its home configuration denoted by A.

2.7 Tree of shortest paths

A question is how the local planner finds a candidate path in an implicitly represented graph. Common for all graph search algorithms like Dijkstra's and A* [17] is that they visit nodes one by one and insert them into a tree whose root is the start node. The tree contains the shortest path from each of the visited nodes to the root. New nodes are visited until the goal node is reached. Then the shortest path can be found by tracing the way back to the root.

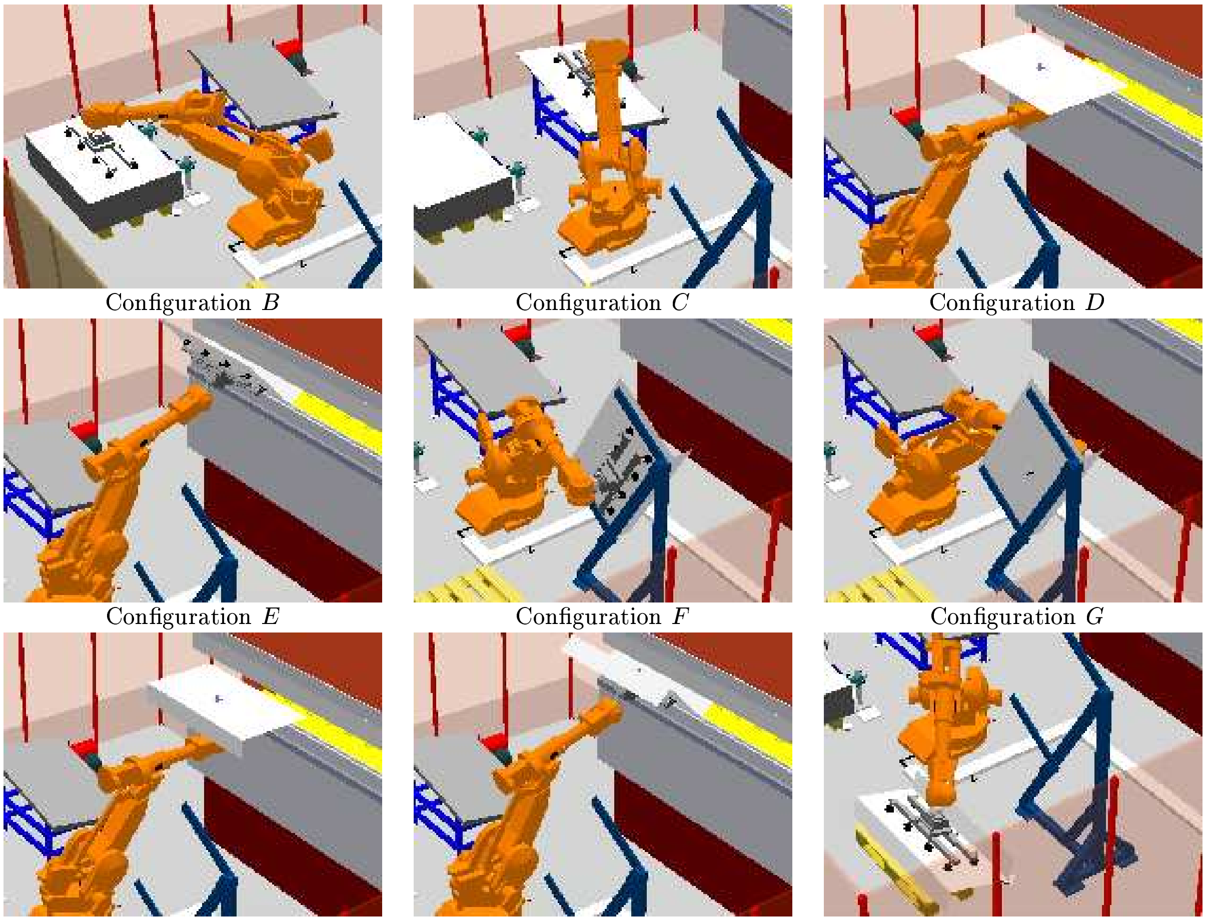
Our local planner uses the A* algorithm. Unfortunately one cannot avoid an explicit representation of the tree, which in the worst case can contain every node in the graph. The iterative behavior of the local planner makes the candidate paths longer and longer, so the tree grows over larger and larger portion of $\mathcal{G}'$. However, what seems to be a problem is also a solution; in each iteration one node is deleted, so the growth will be impeded as well. In practice only a fraction of the nodes will be visited.

Each time a candidate path is rejected, the tree of shortest paths becomes invalid because a deleted node is contained in the tree. Instead of rebuilding the entire tree, we can update the part of the tree that is affected. Only the subtree whose root is the deleted node needs to be updated, which saves much time.

3 Experimental Results

The planner described in previous section has been implemented in C++ as a plug-in module to RobotStudio[1] – a simulation and off-line programming software running under Windows NT. The collision checks are handled internally in RobotStudio. The experiments have been run on a PC with a 400 MHz Pentium II processor and 512 MB RAM.

[1]RobotStudio is developed by ABB Robotics, Göteborg, Sweden.



Configuration B Configuration C Configuration D

Configuration E Configuration F Configuration G

Configuration H Configuration I Configuration J

Figure 3: Configurations B to J used in the experiments.

Our test example is a part of a manufacturing process in which an ABB 4400 robot is tending press breaking. In this particular case, plane sheets of metal are picked from a pallet, bent twice by the hydraulic press shown in Figure 2, and then placed at another pallet.

Ten different configurations denoted A to J are shown in Figures 2 and 3. These are used as either initial or goal configurations in eight planning tasks, denoted for example $A \to B$, where A is the initial configuration and B is the goal configuration. This setting is identical with the experiments reported in [3] and we have chosen the parameters $\{n_i\}_{i=1}^{d}$ to conform to the parameters in [3]. Our aim is to compare our planner with Lazy PRM.

Table 1 shows the total number of collision checks and the total planning time required to solve each of the tasks. The number of collision checks on the solution path and the time spent on collision checking are also shown. Within parenthesis are the corresponding results for Lazy PRM reported in [3]. Since Lazy PRM is a probabilistic planner, we use the average values over 20 consecutive runs.

In Table 1 we find a significant difference between the planners. The number of collision checks to solve all eight tasks is clearly reduced (1671 compared with 2730). This also reduces the planning time to the same extent (139 seconds compared with 289 seconds).

To be a randomized planner, Lazy PRM is very efficient since as much as 26% of the collision checks are on the solution path. Our new planner is even more efficient, 43% are on the solution path. Moreover, our planner spends less time building and searching in the graph. As much as 93% of the planning time is spent on collision checking, whereas Lazy PRM spends 80% on collision checking.

The underlying principle that makes the new planner better in many situations is that the nodes are distributed on a grid. Sparse regions and dense clusters certainly occur if the nodes are randomly distributed, but on a grid (or a subgrid) these can be avoided, or even controlled to improve planning.

4 Summary

We have presented a resolution complete path planner based on an implicit multi-resolution graph in $\mathcal{C}$. To minimize collision checking, and thereby increase per-

Task	Collision checks				Planning time			
	Total		On returned path		Total		Collision checking	
$A \to B$	127	(92)	92	(78)	7.6	(12.7)	7.4	(6.1)
$B \to C$	63	(166)	59	(60)	5.0	(20.2)	4.9	(13.3)
$C \to D$	364	(445)	103	(86)	32.6	(36.8)	27.8	(35.4)
$E \to F$	282	(499)	80	(82)	24.3	(52.0)	23.5	(42.3)
$F \to G$	333	(412)	131	(121)	26.4	(45.7)	25.9	(38.3)
$G \to H$	67	(293)	54	(80)	5.0	(32.5)	4.9	(24.7)
$I \to J$	337	(682)	111	(114)	31.5	(71.0)	28.8	(59.8)
$J \to A$	98	(142)	87	(82)	6.8	(18.2)	6.4	(11.6)
Sum:	1671	(2730)	717	(704)	139.2	(289.0)	129.8	(231.6)
			43%	(26%)			93%	(80%)

Table 1: Performance data for our planner in a test environment. Within parenthesis are corresponding data for Lazy PRM based on 20 consecutive runs for each task.

formance, a scheme for lazy evaluation is applied to the graph.

The algorithm is particularly useful in high dimensional, relatively uncluttered configuration spaces, especially when collision checking is an expensive operation. Single queries are handled very quickly since no preprocessing is required. The planner uses only a boolean collision checker which makes the planner easy to apply even in situations with complex geometry, like in industry. A comparison with Lazy PRM shows significant improvements in terms of planning time.

Acknowledgments

The author would like to thank Bo Johansson for valuable supervision and support, Lydia Kavraki for discussions giving insights in the field of path planning, and ABB Robotics for initiating the project and for providing suitable software.

References

[1] N.M. Amato, O.B. Bayazit, L.K. Dale, C. Jones, and D. Vallejo. OBPRM: An obstacle-based PRM for 3D workspaces. In P. K. Agarwal, L. E. Kavraki, and M. Mason, editors, *Robotics: The Algorithmic Perspective*, pages 630–637. AK Peters, 1998.

[2] J. Barraquand and J.C. Latombe. Robot motion planning: A distributed representation approach. *Int. J. of Rob. Research*, 10:628–649, 1991.

[3] R. Bohlin and L.E. Kavraki. Path planning using lazy PRM. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 2000.

[4] M.S. Branicky, S.M LaValle, K. Olson, and L. Yang. Quasi-randomized path planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 2001.

[5] J.F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.

[6] P. C. Chen and Y. K. Hwang. SANDROS:a dynamic graph search algorithm for motion planning. *IEEE Tr. on Rob. & Aut.*, 14(3):390–403, 1998.

[7] K. Gupta and A. P. del Pobil. *Practical Motion Planning in Robotics.* John Wiley, West Sussex, England, 1998.

[8] D. Henrich, C. Wurll, and H. Wörn. On-line path planning with optimal c-space discretization. In *Proc. IEEE/RSJ Int. Conf. on Int. Rob. and Syst.*, 1998.

[9] T. Horsch, F. Schwarz, and H. Tolle. Motion planning for many degrees of freedom - random reflections at C-space obstacles. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 1994.

[10] D. Hsu, J. C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 1997.

[11] Y.K. Hwang and N. Ahuja. Gross motion planning - a survey. *ACM Comp. Surveys*, 24(3):219–291, 1992.

[12] L.E. Kavraki and J.C. Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 1994.

[13] L.E. Kavraki, P. Švestka, J.C. Latombe, and M. Overmars. Probabilistic roadmaps for fast path planning in high dimensional configuration spaces. *IEEE Tr. on Rob. & Aut.*, 12:566–580, 1996.

[14] K. Kondo. Motion planning with six degrees of freedom by multistrategic bidirectional heuristic free-space enumeration. *IEEE Tr. on Rob. & Aut.*, 7(3):267–277, 1991.

[15] J.C. Latombe. *Robot Motion Planning.* Kluwer, Boston, MA, 1991.

[16] S.M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 1999.

[17] G.F. Luger and W.A. Stubblefield. *Artificial intelligence and the design of expert systems.* Benjamin/Cummings, Redwood City, CA, 1989.

[18] E. Mazer, J.M. Ahuactzin, and P. Bessière. The Ariadne's clew algorithm. *J. of Art. Intelligence Research*, 9:295–316, 1998.

[19] C.L. Nielsen and L.E. Kavraki. A two level fuzzy PRM for manipulation planning. In *Proc. IEEE/RSJ Int. Conf. on Int. Rob. and Syst.*, 2000.

[20] M. Overmars and P. Švestka. A probabilistic learning approach to motion planning. In K.Y. Goldberg, D. Halperin, J.C. Latombe, and R.H. Wilson, editors, *Algorithmic Foundations of Robotics*, pages 19–37. A K Peters, 1995.