A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking

Gildardo Sánchez[1] and Jean-Claude Latombe[2]

[1] ITESM, Campus Cuernavaca, Cuernavaca, México
[2] Computer Science Department, Stanford University, Stanford, CA, USA

Abstract. This paper describes a new probabilistic roadmap (PRM) path planner that is: (1) single-query – instead of pre-computing a roadmap covering the entire free space, it uses the two input query configurations as seeds to explore as little space as possible; (2) bi-directional – it explores the robot's free space by concurrently building a roadmap made of two trees rooted at the query configurations; (3) adaptive – it makes longer steps in opened areas of the free space and shorter steps in cluttered areas; and (4) lazy in checking collision – it delays collision tests along the edges of the roadmap until they are absolutely needed. Experimental results show that this combination of techniques drastically reduces planning times, making it possible to handle difficult problems, including multi-robot problems in geometrically complex environments.

1 Introduction

Probabilistic roadmaps (PRM) have proven to be an effective tool to solve path-planning problems with many degrees of freedom (dofs) [8–10] and/or complex admissibility constraints (e.g., kinodynamic, stability, and visibility constraints) [5,8,11,12]. A PRM planner samples the configuration space at random and retains the collision-free points as *milestones*. It connects pairs of milestones by simple paths (straight segments in configuration space) and retains the collision-free ones as *local paths*. The milestones and local paths form the *probabilistic roadmap*. The motivation is that, while it is often impractical to compute an explicit representation of the collision-free subset of a configuration space (the *free space*), algorithms exist that efficiently test if a given configuration or a local path is collision-free [2,6]. Under some assumptions, the probability that a PRM planner finds a collision-free path, if one exists, goes to 1 exponentially in the number of milestones [7,8]. Hence, random sampling provides a convenient incremental path-planning scheme.

PRM planners spend most of their time performing collision checks. Several approaches are possible to reduce the overall cost of collision checking:

- *Design faster collision checkers.* However, several efficient algorithms already exist. Among them, hierarchical algorithms pre-compute a multi-level bounding approximation of every object in an environment [6,14]. For each collision query, they use this approximation to quickly rule out

large subsets of the objects that cannot collide. They scale up well to complex environments [7].

- *Design sampling strategies yielding smaller roadmaps.* For example, the strategy in [9] produces a first roadmap by sampling configuration space uniformly; next, it picks additional milestones in neighborhoods of existing milestones with no or few connections to the rest of the roadmap. Other strategies generate a greater density of milestones near the boundary of the free space, as the connectivity of narrow regions is more difficult to capture than that of wide-open regions [1,4].
- *Delay collision checks until they are absolutely needed.* The planner in [3] first generates a network by distributing points at random in configuration space. It initially assumes that all points and connections between them are collision-free. It then computes the shortest path in this network between two query configurations and tests it for collision. If a collision is detected, the node and/or segment where it occurs are erased, and a new shortest path is computed and tested; and so on.

We think that delaying collision tests is a promising approach, but its potential has only been partially exploited in [3]. One must decide in advance how large the network should be. If it is too coarse, it may fail to contain a solution path. But, if it is too dense, time will be wasted checking similar paths for collision. The focus on shortest paths may be costly when obstacles force the robot to take long detours.

In this paper, we present a new PRM planner – called SBL, for Single-query, Bi-directional, Lazy in collision checking – that tries to better exploit delayed collision checking, in particular by combining it with single-query, bi-directional, and adaptive sampling techniques, some of which were introduced in [7,8]. SBL incrementally constructs a network of milestones made of two trees rooted at the query configurations, hence focusing its attention to the subset of the free space that is reachable from these configurations. It also locally adjusts the sampling resolution to take longer steps in opened regions of the free space and shorter ones in narrow regions. It does not immediately test connections between milestones for collision. Only when a sequence of milestones joining the two query configurations is found, the connections between milestones along this path are tested. This test is performed at successive points ordered according to their likelihood of revealing a collision. No time is wasted testing connections that are not on a candidate path and relatively little time is spent checking connections that are not collision-free. On a 1-GHz Pentium III processor, the planner reliably solves problems with 6-dof robots in times ranging from a small fraction of a second to a few seconds. Comparison with a similar planner using a traditional collision-checking strategy shows that lazy collision checking cuts planning times by factors from 4 to 40 in the environments of Fig. 1. SBL also solves multi-robot problems reliably and efficiently, like the one in Fig. 5 (36 dofs).

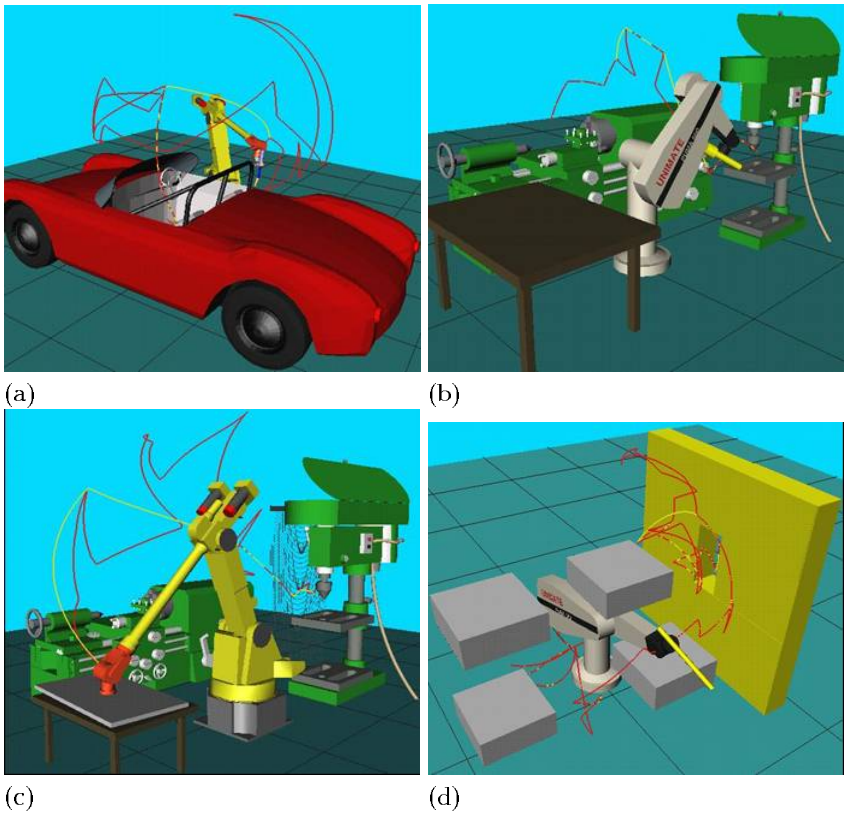

Fig. 1. Path planning environments

2 Definitions and Notations

Let C denote the configuration space of a robot and $F \subseteq C$ its free space. We normalize the range of values of each dof to be [0,1] and we represent C as $[0,1]^n$, where n is the number of dofs of the robot. We define a metric d over C. For any $q \in C$, the neighborhood of q of radius r is the subset $B(q,r) = \{q' \in C \,|\, d(q,q') < r\}$. With $d = L_\infty$ – the metric used by SBL – it is an n-D cube.

No explicit geometric representation of F is computed. Instead, given any configuration $q \in C$, a collision checker returns whether $q \in F$. A path τ in C is considered collision-free if a series of points on τ, such that every two successive points are closer apart than some ε, are all collision-free. A rigorous test (eliminating the need for ε) is possible when a distance-computation algorithm is used instead of a pure collision checker [2].

A path-planning *query* is defined by two *query configurations*, q_{init} and q_{goal}. If these configurations lie in the same connected component of F, the planner should return a collision-free path between them; otherwise, it should indicate that no such path exists. There are two main classes of PRM planners: *multi-query* and *single-query*. A multi-query planner pre-computes a roadmap, which it later uses to process multiple queries [9,10]. To deal with any possible query, the roadmap must be distributed over the entire free space. Instead, a single-query planner computes a new roadmap for each query [7]. The less space it explores to find a solution path, the better. Single-query planners are more suitable in environments with frequent changes.

A single-query planner either grows one tree of milestones from either q_{init} or q_{goal}, until a connection is found with the other query configuration (single-directional sampling), or grows two trees concurrently, respectively rooted at q_{init} and q_{goal}, until a connection is found between the two trees (bi-directional sampling) [8]. In both cases, milestones are iteratively added to the roadmap. Each new milestone m' is selected in a neighborhood of a milestone m already installed in a tree T, and is connected to m by a local path (hence, m' becomes a child of m in T). Bi-directional planners are usually more efficient than single-directional ones.

SBL is a single-query, bi-directional PRM planner. Unlike previous such planners, it does not immediately test the connections between milestones for collision. Therefore, rather than referring to the connection between two adjacent nodes in a roadmap tree as a *local path*, we will call it a *segment*.

3 Experimental Foundations

The design of SBL was suggested by experiments that we performed with the single-query PRM planner described in [8]. To study the impact of collision checking on the running time, we modified the planner's code by removing collision checks for connections between milestones. As we expected, the planner was faster by two to three orders of magnitude, but surprisingly a significant fraction of the generated paths were actually collision-free.

Every segment created by the planner of [8] is relatively short. Thus, the above observation suggested that if two configurations picked at random are both collision-free and close to each other, then the straight-line segment between them has high prior probability of being collision-free. To verify this analysis, we generated 10,000 segments at random with L_∞ lengths uniformly distributed between 0 and 1 (recall that the L_∞ diameter of C is 1). This was done by picking 100 collision-free configurations in C uniformly at random and connecting each such configuration q to 100 additional collision-free configurations obtained by randomly sampling neighborhoods of q of different radii. We then tested each of the 10,000 segments for collision. The chart of Fig. 2 (generated for the environment of Fig. 1(a)) displays the ratio of the number of segments that tested collision-free, as a function of the lengths

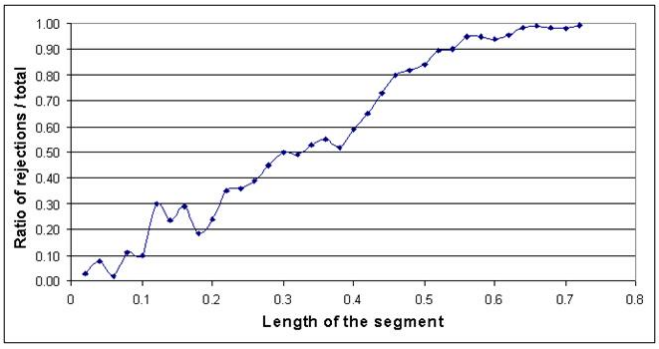

Fig. 2. Collision ratios

of these segments. Here, a segment shorter than 0.25 has probability greater than 0.6 of being collision-free. Similar charts were obtained with the other environments of Fig. 1. There is a simple explanation for this result. Since the robot and the obstacles are "thick" along all or most directions, the obstacle regions in C are also thick in most directions. Hence, a short colliding segment with collision-free endpoints is necessarily almost tangential to an obstacle region in C, an event that has small probability.

The above test and other tests led to making the following observations:

- Most local paths in a probabilistic roadmap are not on the final path. Using the planner of [8] on the examples of Fig. 1, we measured that the ratio of milestones on the final path varies between 0.1 and 0.001.
- The test of a connection is most expensive when it is actually collision-free. Indeed, the test ends as soon as a collision is detected, but is carried down to the finest resolution when there is no collision.
- A short connection between two milestones has high prior probability of being collision-free. Thus, testing connections early is likely to be both useless and expensive.
- If a connection between milestones is colliding, its midpoint has high probability to be in collision; hence, this point should be tested next (a choice that can be applied recursively).

SBL's lazy collision-checking strategy derives from these observations.

4 Description of SBL

SBL is given two parameters: s – the maximum number of milestones that it is allowed to generate – and ρ – a distance threshold. Two configurations are considered "close" to one another if their L_∞ distance is less than ρ. In our implementation, ρ is typically set between 0.1 and 0.3.

4.1 Overall algorithm

Algorithm PLANNER(q_{init},q_{goal})
1. Install q_{init} and q_{goal} as the roots of T_{init} and T_{goal}, respectively
2. Repeat s times
   2.1. EXPAND-TREE
   2.2. $\tau \leftarrow$ CONNECT-TREES
   2.3. If $\tau \neq$ *nil* then return τ
3. Return *failure*

The planner builds two trees of milestones, T_{init} and T_{goal}, respectively rooted at q_{init} and q_{goal}. At each loop of Step 2, EXPAND-TREE adds a milestone to one of the two trees, while CONNECT-TREES connects the two trees. The planner returns *failure* if it has not found a solution path after s iterations. If the planner returns *failure*, either no collision-free path exists between q_{init} and q_{goal}, or the planner actually failed to find one.

4.2 Tree expansion

Algorithm EXPAND-TREE
1. Pick T to be either T_{init}, or T_{goal}, each with probability 1/2
2. Repeat until a new milestone q has been generated
   2.1. Pick a milestone m from T at random, with probability $\pi(m)$
   2.2. For $i = 1, 2, ...$ until a new milestone q has been generated
      2.2.1. Pick a configuration q uniformly at random from $B(m, \rho/i)$
      2.2.2. If q is collision-free then install it in T as a child of m

---

Each expansion of the roadmap consists of adding a milestone to one of the two trees. The algorithm first selects the tree T to expand. Next, a milestone m is picked from T with probability $\pi(m)$ inverse to the current density of milestones of T around m. (Implementation details will be given in Subsection 4.5.) Finally, a collision-free configuration q is picked at distance less than ρ from m. This configuration is the new milestone. The use of the probability distribution $\pi(m)$ at Step 2.1 was introduced in [7] to avoid over-sampling regions of F. It guarantees that the distribution of milestones eventually diffuses through the subsets of F reachable from q_{init} and q_{goal}. This condition is needed to prove that the planner finds a path with high probability, when one exists [7,8]. The alternation between the two trees prevents any tree from eventually growing much bigger than the other, as the advantages of bi-directional sampling would then be lost.

Step 2.2 implements an adaptive sampling strategies, by selecting a series of milestone candidates, at random, from successively smaller neighborhoods of m, starting with a neighborhood of radius ρ. When a candidate q tests collision-free, it is retained as the new milestone. On the average, the jump

from m to q is greater in wide-open regions of F than in narrow regions. Note that the collision test of the segment from m to q is not done here; it will be done later if and when this segment belongs to a candidate path.

4.3 Tree connection

Algorithm CONNECT-TREES
1. $m \leftarrow$ most recently created milestone
2. $m' \leftarrow$ closest milestone to m in the tree not containing m
3. If $d(m, m') < \rho$ then
   3.1. Connect m and m' by a bridge w
   3.2. $\tau \leftarrow$ path connecting q_{init} and q_{goal}
   3.3. Return TEST-PATH(τ)
4. Return *nil*

Let m now denote the milestone that was just added by EXPAND-TREE. Let m' be the closest milestone to m in the other tree. The two trees are connected by a segment, called a *bridge*, joining m and m' if these two milestones are less than ρ apart. The bridge creates a path τ joining q_{init} and q_{goal} in the roadmap. The segments along τ, including the bridge, are now tested for collision. TEST-PATH returns *nil* if it detects a collision.

4.4 Path testing

SBL associates a collision-check index $\kappa(u)$ with each segment u between milestones (including the bridge). This index takes an integer value indicating the resolution at which u has already been tested. If $\kappa(u) = 0$, then only the two endpoints of u (which are both milestones) have been tested collision-free. If $\kappa(u) = 1$, then the two endpoints and the midpoint of u have been tested collision-free. More generally, for any $\kappa(u)$, $2^{\kappa(u)} + 1$ equally distant points of u have been tested collision-free. Let $\lambda(u)$ denote the length of u. If $2^{-\kappa(u)}\lambda(u) < \varepsilon$, then u is marked *safe*. The index of every new segment is initialized to 0.

Let $\sigma(u, j)$ designate the set of points in u that must have already been tested collision-free in order for $\kappa(u)$ to have the value j. The algorithm TEST-SEGMENT(u) increments $\kappa(u)$ by 1:

Algorithm TEST-SEGMENT(u)
1. $j \leftarrow \kappa(u)$
2. For every $q \in \sigma(u, j+1) \backslash \sigma(u, j)$ if q is in collision then return *collision*
3. If $2^{-(j+1)}\lambda(u) < \varepsilon$ then mark u *safe*, else $\kappa(u) \leftarrow j + 1$

For every segment u that is not marked *safe*, the current value of $2^{-\kappa(u)}\lambda(u)$ is cached in the data structure representing u. The smaller this value, the greater the probability that u is collision-free.

Let $u_1, u_2, ..., u_p$ denote all the segments in the path τ that are not already marked *safe*. TEST-PATH(τ) maintains a priority queue U of these segments sorted in decreasing order of $2^{-\kappa(u_i)}\lambda(u_i)$ ($i = 1$ to p). A similar technique has been previously used in [13].

Algorithm TEST-PATH(τ)
1. While U is not empty do
   1.1. $u \leftarrow$ extract(U)
   1.2. If TEST-SEGMENT(u) = *collision* then
      1.2.1. Remove u from the roadmap
      1.2.2. Return *nil*
   1.3. If u is not marked *safe* then re-insert u into U
2. Return τ

Each loop of Step 1 results in increasing the index of the segment u that is in first position in U. This segment is first removed from U. It is later re-inserted into U if TEST-SEGMENT(u) at Step 1.2 neither detects a collision, nor marks u *safe*. If u is re-inserted into U, it may not be in first position, since the quantity $2^{-\kappa(u)}\lambda(u)$ has been divided by 2. TEST-PATH terminates when a collision is detected – then the colliding segment is removed from the roadmap – or when all segments have been marked *safe* (i.e., U is empty) – then the path τ is returned.

The removal of a segment u disconnects again the roadmap into two trees. If u is the bridge that CONNECT-TREES created to connect the two trees, then the two trees return to their previous state (except for the collision-check indices of some segments, whose values may have increased). Otherwise, the removal of u results in a transfer of milestones from one tree to the other. Assume that u is in T_{goal}, as illustrated in Fig. 3(a), where $w \neq u$ denotes the bridge added by CONNECT-TREES. The milestones $m_1, ..., m_r$ between u and w ($r = 3$ in Fig. 3) and their children in T_{goal} are transferred to T_{init} as shown in Fig. 3(b). The parent-child connections between the transferred milestones remain the same, except those between $m_1, ..., m_r$, which are inverted. So, no milestone is ever removed from the roadmap and the collision-checking work done along the segments is saved in their indices. If one of these segments later lies on another candidate path, then the tests previously done are not repeated.

4.5 Implementation details

SBL's collision checker is PQP [6]. Each obstacle and robot link is described by a collection of triangles representing its surface. PQP pre-computes a bounding hierarchical representation of each object using oriented-bounding boxes. No other pre-computation is done by the planner.

The planner spatially indexes every milestone of T_{init} (resp. T_{goal}) in an h-dimensional ($h = 2$ or 3) array A_{init} (resp. A_{goal}). Both arrays partition

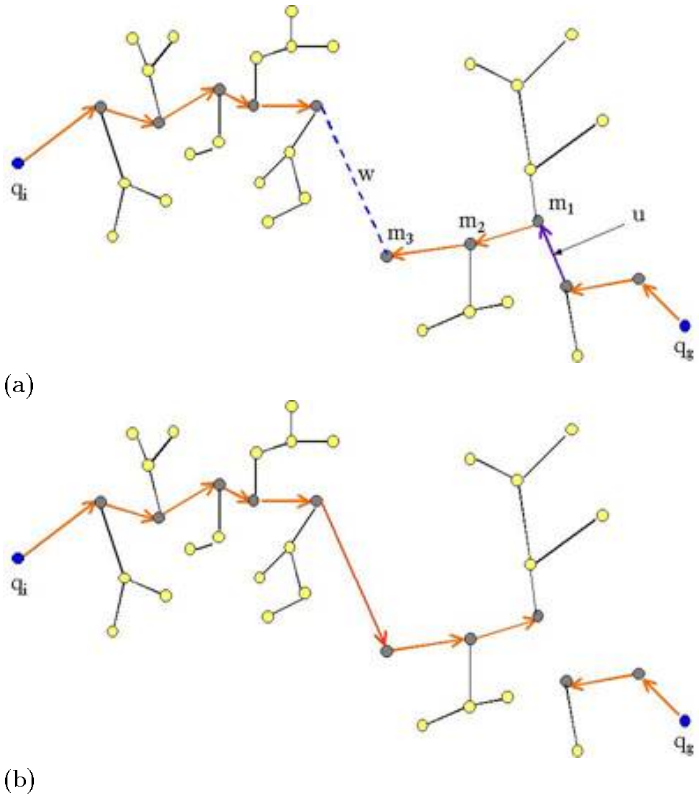

Fig. 3. Transfer of milestones from one tree to the other

the subspace defined by h dimensions of C (in our implementaion, $h = 2$) into the same grid of equally sized cells. Whenever a new milestone q is installed in a tree, the appropriate cell of the corresponding array is updated to contain q. When a milestone is transferred from one tree into the other, the two arrays are updated accordingly. A_{init} and A_{goal} are used at Step 2.1 of EXPAND-TREE, where we pick a milestone m from one tree T with a probability distribution $\pi(m)$. Rather than maintaining the density of samples around each milestone, we do the following. Assume that $T = T_{\text{init}}$. We first pick a non-empty cell of A_{init}, then a milestone from this cell. Hence, the probability to pick a certain milestone is greater if this milestone lies in a cell of A_{init} containing fewer milestones. This technique is fast and results in a good diffusion of milestones in F along the h selected dimensions. To ensure diffusion along all dimensions of C, we periodically change the h dimensions. Each change requires re-constructing the arrays A_{init} and A_{goal}, but the total cost of this operation is negligible relative to collision tests.

Step 2 of CONNECT-TREES also uses A_{init} and A_{goal} to identify the milestone m' that will be connected to the newly added milestone m. Our

implementation of CONNECT-TREES tries two connections: first, instead of selecting m' as the closest milestone to m in the other tree, it picks m' to be the closest milestones in the same cell as m, but in the other array (m and m' are then only guaranteed to be close to each other along h dimensions); then, it picks m' uniformly at random in the other tree. Our experiments have shown that this technique is faster on average than connecting m to the closest milestone. (The "closest-milestone" heuristic often delays the finding of some easy connections.)

Finally, we added a simple path optimizer to SBL to remove blatant jerks from paths. This optimizer takes a path τ as input and performs the following operation several times: pick two points q and q' in τ at random and, if the straight-line segment connecting them tests collision-free, replace the portion of τ between q and q' by this segment.

5 Experimental Results

SBL is written in C++. The running times reported below were obtained on a 1-GHz Pentium III processor with 1 GB of main memory running Linux. The distance threshold ρ was set to 0.15 and the resolution ε to 0.01. Each array A_{init} and A_{goal} had size 10×10. The two dimensions of these arrays were changed every 50 milestones.

Fig. 1 displays some of the single-robot examples we used to test SBL. In each example, the robot is a 6-dof arm; the dark curve is traced by the center-point of its end-effector for one non-optimized path generated by SBL and the light curve is defined in the same way for the optimized path. The numbers of triangles in the robot and the obstacles, n_{rob} and n_{obs}, are indicated in Table 1. The geometrically simpler example of Fig. 1(d) was intended to test SBL when the free space contains narrow passages, a notorious difficulty for PRM planners. The time used by PQP to pre-compute the bounding hierarchies goes from 0.19s for the environment with the fewest triangles (Fig. 1(d)) to 3.9s for the environment with the most triangles (Fig. 1(c)). It is not included in the running times of SBL given below.

Table 1. Number of triangles in robots and obstacles

1(a)		1(b)		1(c)		1(d)	
n_{rob}	n_{obs}	n_{rob}	n_{obs}	n_{rob}	n_{obs}	n_{rob}	n_{obs}
5,000	21,000	3,000	50,000	5,000	83,000	3,000	50

5.1 Basic performance evaluation

Table 2 gives statistics – average running time, standard deviation of running time, average time spent in collision checking, average number of milestones

in final roadmaps, average number of milestones on generated paths, average number of collision checks, average number of checks on generated paths – over 100 runs of SBL on each of the four examples of Fig. 1. In all 400 runs, SBL found a path in small amount of time; there was no failure (the maximal number of milestones s was set to 10,000). A large fraction of the collision checks were made on the solution paths. The running times of Table 2 do not include path optimization, which in all cases took an additional 0.1 to 0.2s.

Table 2. Results on the examples of Fig. 1 (times are in seconds)

ex.	time	std	cc-t	mil	mil-p	#cc	#cc-p
1(a)	0.60	0.38	0.58	159	13	1483	342
1(b)	0.17	0.07	0.17	33	10	406	124
1(c)	4.42	1.86	4.17	1405	24	7267	277
1(d)	6.99	3.55	6.30	4160	44	12228	447

The chart of Fig. 4 was generated by running SBL many times on the example of Fig. 1(c) with increasing values of the maximum number s of milestones, from very small ones to larger ones (horizontal axis). For each value of s, we ran SBL 200 times with different seeds, and we counted the number of failures (vertical axis). When s is very small, SBL fails consistently. When it is suficiently large, its sucess rate is 100%. The transition between consistent failure and consistent success is quite fast, which is coherent with the theoretical result that a PRM planner has fast convergence rate in the number of milestones [7].

Fig. 4. Experimental convergence rate of SBL on the example of Fig. 1(c)

On several examples, we tried different values of ρ between 0.1 and 0.3, as well as indexing arrays of resolutions other than 10×10, including 3-D arrays, but performance results were not significantly different.

5.2 Comparative performance evaluation

To assess the efficiency of our lazy collision-checking strategy, we implemented a version of the planner that fully tests every segment between a milestone and a milestone candidate before inserting the later in the roadmap. This planner is similar to the one presented in [8]. Note, however, that our two planners do not exactly generate the same milestones, even when they use the same seed for the random number generator. Indeed, while SBL considers any collision-free configuration q picked in the neighborhood of a milestone m as a new milestone (Step 2.2 of EXPAND-TREE), the second planner requires in addition that the segment joining m and q be collision-free. Moreover, in the second planner no milestone is ever transferred from one tree to the other.

Table 3. Results with full-collision-check planner

ex.	time	std	cc-t	mil	mil-p	#cc	#cc-p
1(a)	2.82	3.01	2.81	22	5	7425	173
1(b)	1.03	0.70	1.02	29	9	2440	123
1(c)	18.46	15.34	18.35	771	16	38975	219
1(d)	293.77	122.75	292.40	6737	24	666084	300

Table 3 shows the results obtained with the full-collision-check planner, on the same four examples as above. The maximal number of milestones s was set to 10,000 and the results are statistics over 100 runs. The average running times (and numbers of collision checks) for SBL are smaller than for the full-collision-check planner by factors ranging from slightly over 4 for the problem of Fig. 1(c) to over 40 for the problem of Fig. 1(d). These results cannot be compared to those in [3], where the improvement was measured relative to a *multi-query* planner, which must pre-compute a large roadmap to cover the entire free space.

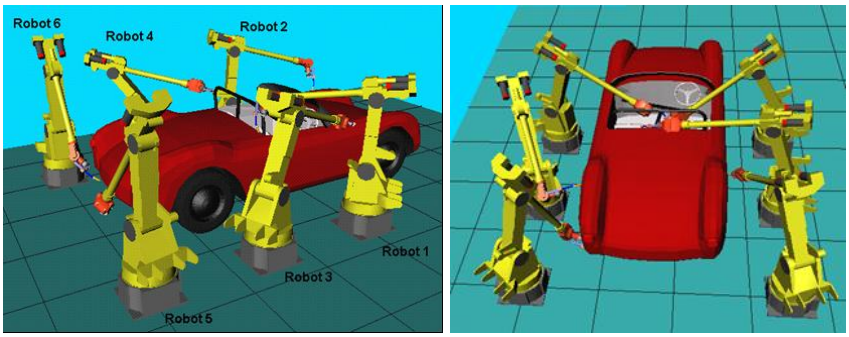

Fig. 5. Multi-robot problem

5.3 Multi-robot examples

We ran SBL on several problems in the environment of Fig. 5, which represents a welding station found in automotive body shops. This environment contains 6 robot arms with 6 dofs each. SBL treats them as a single robot with 36 dofs (centralized planning) and checks collisions between arms.

Table 4. Average running times (in seconds) on 9 multi-robot examples

PI-2	PI-4	PI-6	PII-2.	PII-4	PII-6	PIII-2	PIII-4	PIII-6
0.26	3.97	28.91	0.25	3.94	59.65	2.44	30.82	442.85

Table 4 gives the average running times on 9 problems, over 100 runs for each problem. Fig. 5 shows the initial and goal configurations for the problem named PIII-6. PIII-2 and PIII-4 are the same problem, but reduced to robots 1 and 2, and robots 1 through 4, respectively. The problems PI-2/4/6 and PII-2/4/6 are simpler, with one of the query configurations being the rest configuration of the arms. In all $100 \times 3 \times 3 = 900$ runs, SBL successfully found a path in a satisfactory amount of time. (The maximum number s of milestones for each run was set to 10,000.) The increase in running time when the number of arms goes from 2 to 4 to 6 is caused by both the growth in the number of pairs of bodies that must be tested at each collision-checking operation and by the greater difficulty of the problems due to the constraints imposed by the additional arms upon the motions of the other arms. A more thorough description and analysis of the application of SBL to multi-robot problems can be found in [15,16]

6 Conclusion

This paper shows that a PRM planner combining a lazy collision-checking strategy with single-query, bi-directional, and adaptive sampling techniques can solve path-planning problems of practical interest (i.e., with realistic complexity) in times ranging from fractions of a second to a few seconds for a single or few robots, and from seconds to few minutes for multiple robots. Improvements are still possible. For example, we could use PQP in its distance-computation mode to reduce the number of calls to this program [2].

Our main goal is to extend SBL to facilitate the programming of multi-robot spot-welding stations in automotive body shops. In particular, each robot must perform several welding operations, but the ordering of these locations is not fully specified. Hence, the planner will have to compute an optimized tour of the welding locations. This is a variant of the Traveling Salesman Problem where multiple "salesmen" visit "cities" concurrently and where the distance between any two "cities" is not given and, instead, must be computed by the planner.

Acknowledgements This research was funded in part by grants from General Motors Research and ABB. G. Sánchez's stay at Stanford was partially supported by ITESM (Campus Cuernavaca) and a fellowship from CONACyT. This work has greatly benefited from discussions with H. González-Baños, C. Guestrin, D. Hsu, L. Kavraki, and F. Prinz. PQP was developed in the Computer Science Dept. at the U. of North-Carolina.

References

1. Amato N.M., Bayazit O.B., Dale L.K., Jones C., Vallejo D. (1998) OBPRM: An Obstacle-Based PRM for 3D Workspace. In Agarwal P.K. et al. (eds), Robotics: The Algorithmic Perspective, A.K. Peters, Natick, MA, 155–168.
2. Barraquand J., Kavraki L.E., Latombe J.C., Li T.Y., Motwani R., Raghavan P. (1997) A Random Sampling Scheme for Path Planning. Int. J. of Robotics Research, **16**(6), 759–774.
3. Bohlin R., Kavraki L.E. (2000) Path Planning Using Lazy PRM. Proc. IEEE Int. Conf. Robotics & Autom., San Francisco, CA.
4. Boor V., Overmars M.H., van der Strappen A.F. (1999) The Gaussian Sampling Strategy for Probabilistic Roadmap Planners. Proc. IEEE Int. Conf. Robotics & Autom., Detroit, MI, 1018–1023.
5. Casal A. (2001) Reconfiguration Planning for Modular Self-Reconfigurable Robots. PhD Th., Aeronautics & Astronautics Dept., Stanford U., CA.
6. Gottschalk S., Lin M., Manocha D. (1996) OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. Proc. ACM SIGGRAPH'96, 171–180.
7. Hsu D., Latombe J.C., Motwani R. (1997) Path Planning in Expansive Configuration Spaces. Proc. IEEE Int. Conf. Robotics & Autom., 2719–2726.
8. Hsu D. (2000) Randomized Single-Query Motion Planning in Expansive Spaces. PhD Th., Computer Science Dept., Stanford University, CA.
9. Kavraki L.E. (1994) Random Networks in Configuration Space for Fast Path Planning. PhD Th., Computer Science Dept., Stanford University, CA.
10. Kavraki L.E., Svestka P., Latombe J.C., Overmars, M.H. (1996) Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. IEEE Trans. Robotics & Autom. **12**(4), 566–580.
11. Kindel R. (2001) Motion Planning for Free-Flying Robots in Dynamic and Uncertain Environments. PhD Th., Aeronaut. & Astr. Dept., Stanford U., CA.
12. Kuffner J.J. (1999) Autonomous Agents for Real-Time Animation. PhD Th., Computer Science Dept., Stanford University, CA.
13. Nielsen C., Kavraki L.E. (2000) A Two-Level Fuzzy PRM for Manipulation Planning. Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems.
14. Quinlan S. (1994) Efficient Distance Computation Between Non-Convex Objects. Proc. Int. IEEE Conf. Robotics & Autom., 3324–3329.
15. Sánchez G., Latombe J.C. (2002) Using a PRM Planner to Compare Centralized and Decoupled Planning for Multi-Robot Systems. Proc. IEEE Int. Conf. Robotics & Autom., Washington, D.C.
16. Sánchez G., Latombe J.C. (2002) On Delaying Collision Checking in PRM Planning – Application to Multi-Robot Coordination. To appear in Int. J. of Robotics Research.