Robotica (2015) volume 33, pp. 1491–1506. © Cambridge University Press 2014
doi:10.1017/S0263574714000873

Speeding up probabilistic roadmap planners with locality-sensitive hashing

Mika T. Rantanen* and Martti Juhola

Computer Science, School of Information Sciences, Kalevantie 4, FI-33014, University of Tampere, Tampere, Finland

(Accepted March 9, 2014. First published online: April 9, 2014)

SUMMARY
A crucial part of probabilistic roadmap planners is the nearest neighbor search, which is typically done by exact methods. Unfortunately, searching the neighbors can become a major bottleneck for the performance. This can occur when the roadmap size grows especially in high-dimensional spaces. In this paper, we investigate how well the approximate nearest neighbor searching works with probabilistic roadmap planners. We propose a method that is based on the locality-sensitive hashing and show that it can speed up the construction of the roadmap considerably without reducing the quality of the produced roadmap.

KEYWORDS: Motion planning; Probabilistic roadmaps; Locality-sensitive hashing; Obstacle avoidance; Path planning.

1. Introduction

The motion planning is an important part of robotics (see e.g. refs. [1–3]). The objective is to find a path for a robot from a start location to a goal location. The robot moves in an environment that has obstacles and it must avoid collisions with them. One important application for the motion planning is to guide unmanned vehicles autonomously.[4] Other applications include, among others, computer games, where motion planning can be used to move game characters,[5] and molecular simulations.[6]

The location of the robot is typically represented as a *configuration*, and the *configuration space* $\mathcal{C}$ contains all possible configurations.[7] The *free configuration space* $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ is a set of all configurations where the robot does not collide with the obstacles or itself. The robot can now be thought to be a single point in $\mathcal{C}$ whereas a path between two locations q_1 and q_2 is a continuous function $\tau : [0, 1] \to \mathcal{C}$, where $\tau(0) = q_1$ and $\tau(1) = q_2$. The path is free if it lies totally in $\mathcal{C}_{\text{free}}$.

The dimension of the configuration space depends on the robot. In case of a rigid body robot that moves in a three-dimensional environment without rotations, three parameters are needed to represent the location. Therefore, the configuration space is three-dimensional and the robot has three degrees of freedom (3-DOF). If the robot can rotate itself freely, three additional parameters are needed (roll, pitch, and yaw). In that case, the configuration space is six-dimensional. The robot can also be more complex similar to a robot arm that has many joint angles. It is also possible that the robot consists of multiple rigid bodies and that they form a single multibody robot. In that case the dimensionality of $\mathcal{C}$ is the same as the sum of degrees of freedom of all the bodies.

The general motion planning problem is PSPACE-complete,[8,9] which means that exact algorithms are not useful in practice. Luckily, there are many heuristic methods that can be used. Especially *probabilistic roadmap* (PRM) planners[10–12] have been shown to work well in difficult environments. These planners construct a graph called a *roadmap*, which is a simplified representation of $\mathcal{C}_{\text{free}}$. Its nodes are randomly sampled configurations from $\mathcal{C}_{\text{free}}$ and an edge between two configurations means that there is a simple and free path between them.

When using PRMs, most of the work is done during the construction of the roadmap, and after the roadmap is ready, it can be used to solve different path planning queries quickly. The

* Corresponding author. E-mail: mika.t.rantanen@uta.fi

roadmap is constructed by using information about the static obstacles of an environment. However, PRM planners are versatile and it is possible to use them also in applications that have dynamic environments.[13,14]

While constructing the roadmap, the PRM planner must find a set of the nearest neighbor nodes every time a new node is added to the roadmap. Using exact methods is fast when the roadmap is small and the dimension of the configuration space is low, but it can become a major bottleneck for performance when the roadmap size grows, especially in high-dimensional spaces. Therefore, more efficient methods should be used.

The nearest neighbor search can be accelerated by using approximation methods such as ANN[15] or FLANN.[16] In this paper, we concentrate on *locality-sensitive hashing* (LSH), which has successfully been applied to a variety of applications lately.[17–19] However, as far as we know, LSH has not been used with PRM planners before. In this paper, we propose a simple LSH-based method that can speed up the construction of roadmaps remarkably. We also investigate how the use of this approximation method affects the quality of roadmaps.

Algorithm 1 Constructs the roadmap $G = (V, E)$

```
 1: $V \leftarrow \emptyset$
 2: $E \leftarrow \emptyset$
 3: repeat
 4:     $q \leftarrow$ sampled configuration from $C_{\text{free}}$
 5:     $V \leftarrow V \cup \{q\}$
 6:     $N_q \leftarrow k$ nearest neighbor configurations of $q$ chosen from $V$
 7:     for all $q'$ in $N_q$ do
 8:         if $q$ and $q'$ are not in the same component of the roadmap then
 9:             if local planner finds a path between $q$ and $q'$ then
10:                 $E \leftarrow E \cup \{(q, q')\}$
11: until there are enough configurations in $V$
12: return $G$
```

2. Probabilistic Roadmap Planners

A typical PRM planner is depicted in Algorithm 1. It starts with an empty roadmap graph $G = (V, E)$, where V is a set of configurations and E is a set of edges. The configurations are added to V one by one. Each time a new configuration q is added, a set of neighboring configurations is retrieved for it from V. Then, the algorithm tries to find a free path from q to each of its neighbors with a *local planner*. If the local planner can find a path, an edge between those configurations is added to E.

A good roadmap captures the *connectivity* of the free configuration space as good as possible. It means that if there is a way for a robot to go from a location q_1 to a location q_2 without colliding obstacles, then there should also be a path in G between configurations q_1 and q_2 after those configurations are added to the roadmap with a local planner. If the connectivity is not good, it is possible that q_1 and q_2 belong to different components and there is not a path between them.

In this paper, we are especially interested in searching the neighbors, and the following sections will cover that topic. Here we go shortly through other important parts of PRM planners.

Node sampling. In line 4 of Algorithm 1, a new configuration is sampled from a free configuration space. The simplest way to generate these new configurations is to sample them randomly using uniform distribution. Another simple method is to use a low-discrepancy sequences to produce the samples.[3,20]

In fully random environments, these sampling methods are adequate. However, the environments where the robots move are often more organized and structured. For example, if the robot moves in a house, there are walls and furniture as obstacles, and large empty areas, where the robot can move freely. In these kinds of environments it is possible to use sampling methods that try to bias sampling toward difficult areas.

Many different methods have been suggested to enhance sampling. Some methods try sample nodes near the obstacles,[21–23] whereas others try to sample nodes as far of the obstacles as possible.[24,25] There are also methods that divide the configuration space into different regions and then try to detect the best way to sample each region.[26–28]

Local planner. In line 9, a local planner is used to check whether a path between two configurations is free. One commonly used local planner is a straight-line planner which just interpolates a path between two configurations (see e.g. ref. [12]). The path is then divided into small steps and each step is checked for collisions separately. Other local planners have also been proposed (see e.g. refs. [29, 30]). These more advanced planners can typically find paths that the straight-line planner cannot, but, on the other hand, they usually work slower.

The collision checks that the local planner does are one of the most time- consuming parts of PRM planners. Therefore, it is not feasible to check all possible edges for collision, and instead a set of neighbor nodes are selected in line 6 and a local planner is called just for them. Moreover, if the goal is just to generate a roadmap that captures the connectivity of the free configuration space well, it is not necessary to add cycles to the roadmap. That is why the algorithm checks in line 8 whether a neighbor configuration is already on the same connected component of the roadmap.

Distance metric. In line 6, a set of the nearest neighbor nodes is selected according to a metric which should be such that the distance between two configurations, p and q, reflects the difficulty of connecting them together with a local planner. One possibility would be to measure the volume of the workspace that is swept by a robot when it moves between p and q. When this swept volume is small, also the probability that the robot will collide with obstacles is small. However, it is very difficult and slow to compute the swept volume exactly. Usually the metrics used in motion planning are approximations.[31] Next, we describe one common approach.

If the configuration space is a Cartesian product of n metric spaces and each has been associated with a distance metric $M_1, M_2, \ldots, M_n$, then the approximate distance between two configurations, $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$ can be defined as

$$\mathrm{dist}(p, q) = \sqrt{\sum_{i=1}^{n} w_{M_i} \mathrm{dist}^2_{M_i}(p_i, q_i)},$$

where $w_{M_i} \in \mathbb{R}$ are weight constants. The distance for two points $p_1, p_2 \in \mathbb{R}$ can be defined as

$$\mathrm{dist}_{\mathrm{T}}(p_1, p_2) = |p_1 - p_2|.$$

Special care must be taken when handling rotations to ensure that the shortest distance is selected. This can be achieved for three-dimensional rotations that are represented as two quaternions $h_1, h_2 \in \mathbb{H}$ by defining the distance as

$$\mathrm{dist}_{\mathrm{R}}(h_1, h_2) = \arccos(|h_1 \cdot h_2|).$$

One difficulty is to decide the values for the weight constants. Usually the translational distances are more important than the rotational distances and, for example, the study in ref. [29] supports this.

3. Nearest Neighbor Search

In the nearest neighbor search a set S of points in a metric space M is given and a goal is to find the nearest point in S to a query point $q \in M$. Often one nearest neighbor is not enough but instead k nearest neighbors for q is required. This is also the case with the PRM planners.

The nearest neighbor search is an important part of PRM planners since for each sampled configuration we need to search for its neighbors. Unfortunately, this can become a very time-consuming operation when the roadmap size increases. The neighbor search is one of the bottlenecks of the PRM planners along with the collision checks.

The nearest neighbor search has many important applications also besides the robotics and it has been researched extensively. Many applications require that the nearest neighbor search always returns an exact result. However, there are also many applications where it is not necessary and a good approximation is enough. This is beneficial because while the exact nearest neighbor algorithms can be slow, there are several approximation methods that can work much faster.

For example, the PRM methods themselves are not exact algorithms but still exact nearest neighbor search is typically used in these methods. However, this is likely unnecessary and as we show in this paper, the approximation methods are sufficient.

We consider a case where the roadmap is empty at the beginning and all nodes are added to the roadmap one by one. This way it is possible to construct the roadmap without knowing how large it will be in advance and adding new nodes to the roadmap is easy. Another approach would be that all nodes are generated first and then added to the roadmap at once.

Next, we describe the nearest neighbor methods that we used in our experiments.

3.1. Brute-force search

The simplest way to find k nearest neighbors is to go through all points in S and for each one calculate the distance from the query point q. While iterating through the points, a list of k nearest neighbors found so far is maintained. At the end, the list has an exact result. The brute-force method is slow since it requires that every point is checked. However, this method does not use any additional data structures which could be difficult and slow to maintain. Therefore, it can outperform other exact methods especially in high-dimensional spaces.[32]

3.2. kd-*tree method*

There are several space-partitioning methods that can be used in exact nearest neighbor searches. One of the simplest is the *kd*-tree method that uses a binary tree to store the points.[33,34]

In the *kd*-tree method each node contains a point from S. Every node divides the space into two partitions by a plane that goes through the associated point. Every node is also associated with one of the dimensions of the space M and the dividing plane is defined to be perpendicular to the axis of that dimension. The left subtree of the node contains all the points that lie on one side of the dividing plane and the right subtree contains the rest of the points.

The results in ref. [35] show that the *kd*-trees can dramatically increase the performance of the PRM planners when compared with the brute-force search. However, *kd*-tree and other space-partitioning methods work well only in low-dimensional spaces and they become inefficient when the dimensionality grows large. In matter of fact, the results in ref. [32] show that a simple brute-force method can outperform space-partitioning methods when the number of dimensions grows larger than around 10.

3.3. Locality-sensitive hashing

The LSH can be used to accelerate the search of the nearest neighbors. It is a popular method and has been used successfully in many different applications, for example, in audio and image retrieval (see e.g. refs. [36, 37]). The method does not guarantee that it will find exactly the nearest neighbors, but it will return a very good approximation with a high probability.

The LSH method is based on a hash table which contains several buckets. Each bucket is associated with a unique hash value and a hash function g is used to calculate a hash value for points in metric space M. By using this function, it is possible to store all points in S to the buckets. It should be noted that one bucket should contain several points.

The hashing function g must be selected in such a way that it returns the same value for two points with a high probability if the points are close to each other according to some distance metric. This means that the hash function preserves the locality information, and the points that are near each other are likely to be stored in the same bucket in the hash table.

To search for a set of points that are near to a query point $q \in M$, we first calculate a hash value for q. Then we just retrieve all points from the bucket indicated by this hash value. The hash tables can be implemented to work very efficiently, which means that in practice the searching of these points can be done quickly. Unfortunately, it is not certain that the found points contain the actually nearest neighbor for q.

It is possible to increase the probability that the method finds the nearest point by using multiple hash tables. It means that instead of one hash table we create independently L tables and for each table we use a different hash function. Now new points must be added to every table as shown in

Algorithm 2. To retrieve the nearest points, we must get points from each table and then combine them together as shown in Algorithm 3.

A variety of different hashing functions have been developed (see e.g. refs. [38, 39]). One simple method is to use randomly chosen hyperplanes. Each plane divides the space into two partitions and together they divide the space into several regions. Another method is to use lattices to create regular-shaped regions into the space. In both cases, the hashing function can return the same hash value for each point that lies in the same region.

The problem that remains is to decide how to choose an appropriate hashing function and how to choose the number of buckets and the number of hash tables. The number of buckets is usually linked to the hashing function, but in general the accuracy can be increased by adding the number of hash tables. This, however, makes the method work slower and consume more memory. When choosing a hashing function, it should be remembered that many proposed functions are designed to work only in Euclidean spaces or in some other specific cases, which means that they may not be useful in all applications. In the next section, we describe a centroid-based hashing that is easy to be implemented and can be extended to satisfy the needs of PRM planners.

Algorithm 2 Adds a point q to the hash tables

```
1: for $i = 1, 2, \dots, L$ do
2:     $h \leftarrow$ hash value for $q$ in $i$th hash table
3:     $b \leftarrow$ bucket identified by a value $h$ from $i$th hash table
4:     Add point $q$ to the bucket $b$
```

Algorithm 3 Returns k nearest nodes to a query point q

```
1: $V \leftarrow \emptyset$
2: for $i = 1, 2, \dots, L$ do
3:     $h \leftarrow$ hash value for $q$ in $i$th hash table
4:     $b \leftarrow$ bucket identified by a value $h$ from $i$th hash table
5:     $V' \leftarrow$ all nodes from bucket $b$
6:     $V \leftarrow V \cup V'$
7: return $V$
```

3.4. *Centroid-based hashing*

In centroid-based hashing,[39] a set of c centroids are generated at the beginning. These centroids belong to the space M, and an arbitrary point $q \in M$ can be associated with one of the centroids by calculating the distance from q to each centroid and then selecting the centroid that is the nearest. This essentially divides the space M into c partitions. This division can also be thought to be the Voronoi diagram[40] where each centroid corresponds to one Voronoi cell.

We can now define the hashing function g in such a way that it takes a point, calculates which Voronoi cell it belongs to, and then returns a hash value associated with that cell. By using this hash function, the number of buckets will be the same as the number of centroids.

The centroid-based hashing is illustrated in two-dimensions in Fig. 1. In Figs. 1(a)–(c), three different sets of centroids and corresponding Voronoi cells are shown. The centroids are marked with a small dot, a query point q is shown in the middle as a small circle, and the cell in which q belongs to is marked with a gray color. To retrieve a set of the nearest points for q, it is sufficient to retrieve points from these three marked Voronoi cells. This is shown in Fig. 1(d), in which the point q, as well as the union of the gray Voronoi cells from Figs. 1(a)–(c), is shown.

Next, we propose a simple centroid-based method that works well with PRMs. We address two issues: How to choose the centroids, and how to handle cases where there are only a few nodes in the roadmap.

The first issue is to choose the centroids. The simplest way to do it would be to generate c random points and use them as centroids. This would work but would ignore some information that we have from the environment. We propose a method that takes the static obstacles into account. Our method to initialize the centroids is shown in Algorithm 4. We assume that the roadmap is empty at the

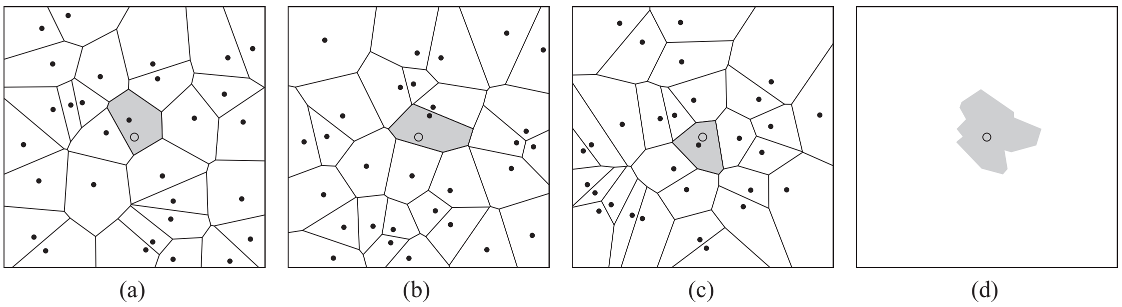


Fig. 1. An example of an LSH. In (a)–(c) the space is divided into several regions. Black dots represent centroids, a small circle represents a query point, and the cell where the query point lies is marked with a gray color. In (d), the marked cells from (a)–(c) are combined.

beginning, which means that it is not possible to use the information about the existing roadmap. Therefore, the method chooses c centroids uniformly at random but requires that all of these must lie in the free configuration space. This method yields good results as we show in our experiments. To distribute the centroids in more evenly manner in the space, it is possible to generate these using some low-discrepancy sequence.

Algorithm 4 Initializes L hash tables with c centroids

```
1: Create $L$ hash tables
2: for $i = 1, 2, \ldots, L$ do
3:     $P \leftarrow \emptyset$
4:     for $j = 1, 2, \ldots, c$ do
5:         $q \leftarrow$ a random free configuration
6:         $P = P \cup \{q\}$
7:     Associate centroids $P$ with $i$th hash table
```

The second issue is to handle cases when the roadmap has only a few nodes. For example, if we want to retrieve k nearest neighbors for a query point and the roadmap has only k nodes, all nodes from the roadmap should be returned. However, the method shown in Algorithm 3 would probably return only a small subset of all nodes because it returns nodes only from some of the buckets. An improved method is shown in Algorithm 5. It immediately returns all nodes if the roadmap has less than k nodes and otherwise ensures that k nearest nodes are always returned.

Algorithm 5 Returns k nearest nodes to a query point q from a set of roadmap nodes R

```
1: if $|R| \leq k$ then
2:     return $R$
3: $V \leftarrow \emptyset$
4: for $i = 1, 2, \ldots, L$ do
5:     $h \leftarrow$ hash value for $q$ in $i$th hash table
6:     $b \leftarrow$ bucket identified by a value $h$ from $i$th hash table
7:     $V' \leftarrow$ all nodes from bucket $b$
8:     $V \leftarrow V \cup V'$
9: if $|V| < k$ then
10:    return $k$ nearest nodes to $q$ from $R$
11: else
12:    return $k$ nearest nodes to $q$ from $V$
```

If both constants c and L are equal to 1, it means that the algorithm works just like the brute-force search. It should be noted in the case of PRM methods that if c is larger than 1, then L must also be larger than 1. Otherwise, the roadmap would be built separately in each Voronoi cell and they would not be connected together. By increasing L there will be overlapping between the cells which helps the roadmap to get connected.

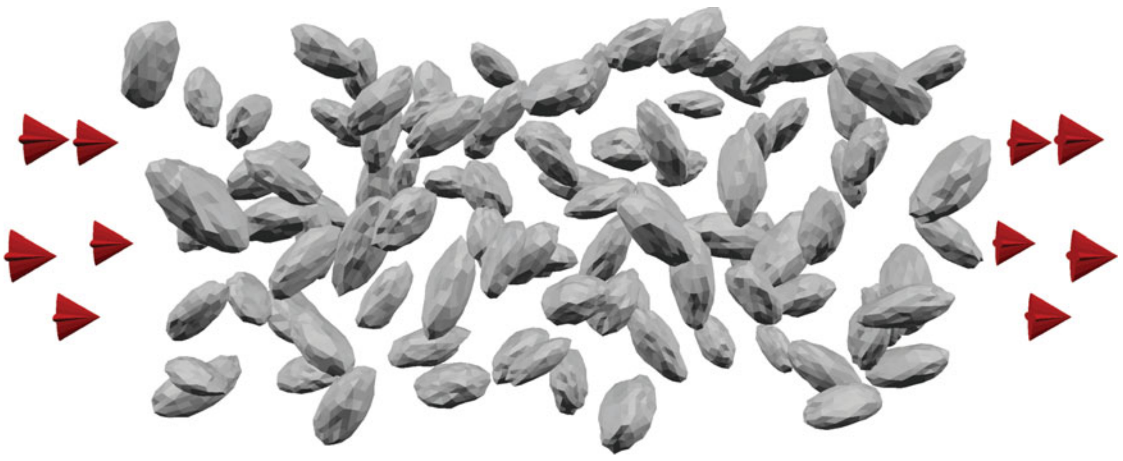


Fig. 2. An asteroids environment. All robot bodies are shown at a start location (left side) and at a goal location (right side).

4. Experiments

In our experiments, we tested how different nearest neighbor methods work with PRM planners. The methods we used were the brute-force search, the *kd*-tree method, and the centroid-based LSH. We used three environments in our tests: an asteroids environment, a house environment, and a wall environment (see Figs. 2–4). In each environment we tested the neighbor methods with three robots. The robots comprised a different number of rigid bodies, which means that for each robot the configuration space had a different dimension. Each method was also tested with two values for *k* which determines how many neighbors are retrieved each time the planner wants to find the nearest neighbors (see line 6 in Algorithm 1). To minimize the effect of randomness we made 1000 test runs for each test case.

Our goal was to measure how much different neighbor methods affected the time required to construct a roadmap and to investigate whether the LSH method reduces the quality of the produced roadmap. We also tested several different parameters for the LSH method to demonstrate their effects.

The size of the roadmap that we built was 20,000 nodes for the asteroids and wall environments, and 30,000 nodes for the house environment. We also had a predefined query that we tried to solve. We measured the used time, number of roadmap components, length of the found path, and how often the method failed to solve the predefined query. The measurements were taken when the construction of the roadmap was finished, at the point when the predefined query had just been solved and when the roadmap had 10,000 nodes.

4.1. Test setup

In every test, the node sampling method, local planner, and used distance metric were the same. Besides the used environments and robots, the only thing that changed between the tests was the used nearest neighbor method. This makes the results that we obtained comparable with each other.

In the node sampling we used a random method where new configurations were sampled from the free configuration space uniformly. The local planner we used was a simple straight-line planner. The distance metric was the sameas described in Section 2. We made these choices because these are probably the simplest methods and require only a minimal number of additional parameters.

All methods were implemented in C++ to the same software framework. Proximity Query Package (PQP) library[41] was used for collision detection. All tests were run on a PC with Intel i7-2600 (3.40 GHz) processor and 8 GB of memory.

4.2. Environments

Each test environment comprised static obstacles and a robot that had several independent rigid body parts. The obstacles and the robot were represented as triangle mesh. Next, we describe the used environments in more detail.

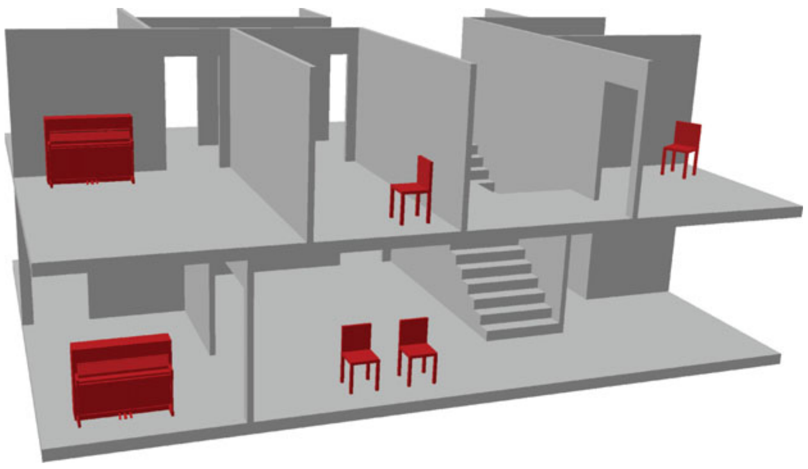


Fig. 3. A house environment (shown without roof and exterior walls). All robot bodies are shown at a start location (downstairs) and at a goal location (upstairs).

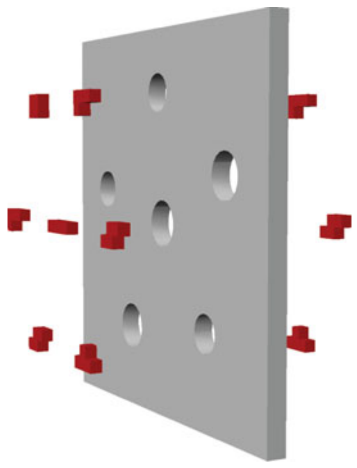


Fig. 4. A wall environment. All robot bodies are shown at a start location (left side) and at a goal location (right side).

Asteroids environment. Obstacles are small asteroids of different sizes and shapes that are spread throughout the space. The environment was generated randomly. A robot consists of several spaceship-shaped parts. There are three different robots: the first has one part, the second has three parts, and the third has five parts. In a predetermined query, the robot must go through the asteroids from one side to the other. The workspace size is 400 × 100 × 100 units. There are 100 asteroids and together they have 32,000 triangles. Each spaceship has 64 triangles. The environment is illustrated in Fig. 2.

House environment. Obstacles of the house consist of walls, floors, stairs, and a roof. There are three different robots: the first is just a piano, the second consists of a piano and one chair, and the third consists of a piano and two chairs. In a predetermined query, all robot parts are located on the first floor and the goal is to find a way to move them to the second floor. The workspace size is 511 × 354 × 220 units. The house consists of 500 triangles in total. The piano has 519 triangles and each of the chairs has 136 triangles. The environment is illustrated in Fig. 3.

Wall environment. An obstacle is a wall that has several circular holes in it and a robot consists of several tetromino-shaped parts. There are three different robots: the first has three parts, the second has five, and the third has seven parts. In a predetermined query, the robot must find a way to go from one side of the wall to the other side. The work space size is 100 × 100 × 100 units. The wall consists of 572 triangles. Each robot part has 20 triangles on average. The environment is illustrated in Fig. 4.

All three environments have their own characteristic properties. The asteroids environment is a cluttered environment that is filled with obstacles. There are not large empty areas anywhere, which makes it difficult for the local planner to find free paths. On the other hand, the wall environment has large empty areas. The only obstacle is the wall that divides the space into two parts. The difficulty is to find a path through the holes in the wall. The house environment can be thought to be a combination of these other environments. It has many obstacles, but also many empty areas where the robot can move freely.

In addition to the obstacles, the environments also differ by their size and shape. The work space where the robot moves is bounded and the wall environment has the smallest working area in contrast to the house environment which has the largest area. In the wall environment the shape of the work space is a cube whereas in other environments it is rectangular cuboid. The shape of the work space can have an impact on the performance of the nearest neighbor methods, such as *kd*-tree, which depends on dividing the space by axis-aligned planes.

5. Results

Results of our experiments are shown in Tables I–III. Except for the success rate, all results are the averaged values of 1000 test runs with standard deviation shown in parentheses. The method column shows the used nearest neighbor method, and in the case of the LSH method, used values for parameters L and c are also shown. The column k shows how many neighbors were retrieved at maximum in each time neighbors were searched for. The robots column shows how many bodies the robot had. The columns marked with text “Path found” show roadmap properties at the time when a predetermined query was solved. The rest of the columns show properties for certain roadmap sizes. It should be noted that once the predetermined query has been solved, the length of the found path does not change even if the number of roadmap nodes grows. This is because the roadmap does not contain cycles and therefore there can be only one path between two nodes.

Since the brute-force method and the *kd*-tree method are exact, it is expected that they give similar results when applied to PRM planners. The only difference should be the time required to build the roadmap. By looking at the results in Tables I–III, we can see that it is the case. The number of roadmap components is approximately the same, as well as the success rates and the lengths of the found paths. The numbers are not exactly the same because of the random nature of PRM planners. This effect can also be seen by looking at standard deviations as they can be quite high at some cases. The results also show that the *kd*-tree method slows down when the dimension of the configuration space increases and that it will eventually be slower than the brute-force method.

By comparing the different methods, it can be seen that LSH speeded up the construction of the roadmap almost in all cases when compared with the exact methods. When looking at the time required to build the whole roadmap, it can be seen that the only exceptions were the cases where there was only one robot body, i.e. the dimension of the configuration space was small. In these cases, the *kd*-tree method was slightly faster. With higher dimensions, the LSH method is considerably faster than either exact method. When looking at the results, it should be remembered that when the roadmap size is very small, our LSH-based method works similar to the brute-force method. Figures 5–7 illustrate how the different methods affect the time used to build the roadmap. In these figures, the used value for parameter k was 100 and the values for LSH parameters L and c were 150 and 20 respectively.

One important aspect that must be taken under consideration when using approximation neighbor methods is the quality of the produced roadmap, which should not differ too much between the exact and approximate methods. We measured the quality of each method by calculating the number of roadmap components in certain roadmap sizes and solving a predetermined path query. On average, all methods should solve the query with the same number of nodes, and the length of the found path should be the same. Moreover, the success rate and the number of components should also be the same and the success rate should increase as the number of roadmap nodes grows. The results in Tables I–III show that LSH can achieve a similar quality as the exact methods with faster running time. However, the quality of the LSH method depends on its parameters.

For comparison, we ran the tests with three different sets of parameters for the LSH method. From these parameters we obtained the best results with 20 hash tables (parameter L) and 150 centroids (parameter c). These parameters gave quite good results in all environments and with different

Table I. Average results of 1000 test runs for asteroids environment, with standard deviations shown in parentheses. SR means success rate. L means the number of hash tables, and c means the number of clusters.

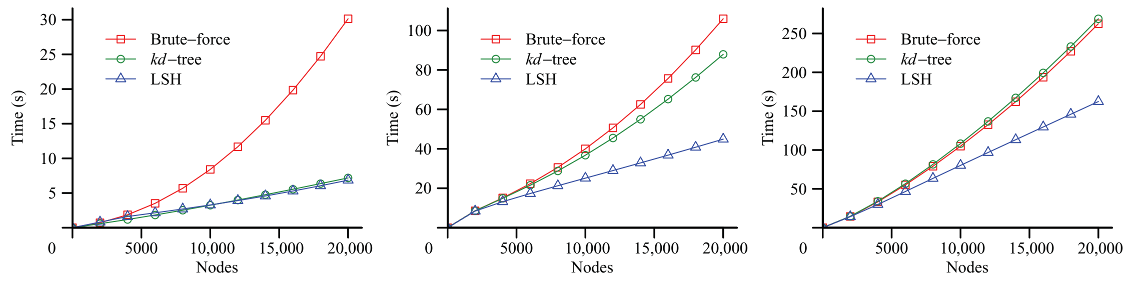
			Path found				10,000 nodes			20,000 nodes		
Method	*k*	Robots	Time (s)	Nodes	Components	Length	Time (s)	Components	SR	Time (s)	Components	SR
Brute-force	50	1	<0.05	41.1 (17.0)	2.3 (1.1)	641.1 (104.1)	8.1 (0.1)	4.1 (1.7)	100	29.3 (0.3)	7.0 (2.3)	100
kd-tree	50	1	<0.05	42.0 (16.9)	2.3 (1.2)	643.9 (107.8)	2.7 (0.1)	4.1 (1.7)	100	5.7 (0.2)	7.0 (2.2)	100
LSH (*L*: 20, *c*: 150)	50	1	<0.05	41.2 (15.9)	2.4 (1.2)	640.9 (104.9)	2.7 (<0.05)	4.0 (1.7)	100	6.2 (0.1)	6.9 (2.2)	100
LSH (*L*: 5, *c*: 150)	50	1	<0.05	42.1 (17.1)	2.3 (1.1)	644.4 (106.8)	2.5 (<0.05)	4.2 (1.7)	100	5.0 (<0.05)	7.0 (2.4)	100
LSH (L: 20, *c*: 300)	50	1	<0.05	40.5 (16.6)	2.3 (1.2)	641.5 (111.1)	3.3 (<0.05)	4.1 (1.7)	100	6.5 (0.1)	7.0 (2.3)	100
Brute-force	100	1	<0.05	41.6 (16.9)	2.3 (1.1)	639.2 (108.0)	8.4 (0.1)	4.1 (1.7)	100	30.1 (0.2)	6.9 (2.3)	100
kd-tree	100	1	<0.05	41.7 (16.2)	2.3 (1.1)	645.5 (104.7)	3.2 (0.1)	4.0 (1.6)	100	7.2 (0.2)	6.9 (2.3)	100
LSH (*L*: 20, *c*: 150)	100	1	<0.05	41.1 (16.2)	2.3 (1.1)	646.3 (106.1)	3.3 (<0.05)	4.0 (1.7)	100	6.9 (0.1)	6.9 (2.3)	100
LSH (*L*: 5, *c*: 150)	100	1	<0.05	40.8 (16.4)	2.2 (1.1)	644.3 (112.5)	3.9 (0.1)	4.1 (1.7)	100	6.5 (0.1)	6.9 (2.3)	100
LSH (*L*: 20, *c*: 300)	100	1	<0.05	42.1 (16.6)	2.3 (1.1)	644.9 (110.4)	5.2 (0.1)	4.1 (1.8)	100	8.4 (0.1)	7.0 (2.4)	100
Brute-force	50	3	2.2 (0.5)	625.8 (162.7)	147.7 (15.0)	3505.3 (922.1)	35.6 (0.3)	290.9 (17.1)	100	100.5 (0.4)	348.3 (19.3)	100
kd-tree	50	3	2.2 (0.5)	616.3 (162.3)	147.9 (15.6)	3517.1 (959.6)	31.7 (1.0)	291.4 (17.3)	100	79.3 (3.6)	350.5 (18.6)	100
LSH (*L*: 20, *c*: 150)	50	3	2.2 (0.5)	621.1 (168.9)	148.3 (16.1)	3555.5 (928.2)	20.4 (0.2)	291.6 (17.0)	100	39.2 (0.4)	350.4 (19.3)	100
LSH (*L*: 5, *c*: 150)	50	3	2.2 (0.5)	622.3 (169.9)	148.9 (16.2)	3473.7 (936.3)	19.7 (0.3)	330.8 (18.7)	100	35.5 (0.4)	388.4 (20.6)	100
LSH (*L*: 20, *c*: 300)	50	3	2.2 (0.5)	616.0 (162.8)	147.9 (15.8)	3451.2 (911.0)	20.8 (0.2)	290.5 (17.6)	100	38.4 (0.3)	349.1 (19.1)	100
Brute-force	100	3	3.3 (0.7)	592.5 (144.2)	140.9 (13.4)	3474.8 (857.7)	40.0 (0.4)	201.7 (14.5)	100	105.9 (0.5)	227.9 (15.1)	100
kd-tree	100	3	3.4 (0.7)	596.7 (155.2)	141.5 (13.3)	3431.2 (915.5)	36.8 (1.0)	201.8 (14.0)	100	87.9 (3.5)	228.2 (15.3)	100
LSH (*L*: 20, *c*: 150)	100	3	3.3 (0.8)	595.1 (158.4)	140.6 (13.4)	3455.7 (932.6)	25.1 (0.3)	202.1 (14.2)	100	44.9 (0.4)	228.2 (14.7)	100
LSH (*L*: 5, *c*: 150)	100	3	3.3 (0.8)	598.5 (156.6)	140.8 (13.3)	3471.4 (929.6)	25.8 (0.3)	240.6 (16.5)	100	42.7 (0.4)	265.8 (16.1)	100
LSH (*L*: 20, *c*: 300)	100	3	3.4 (0.8)	600.6 (156.8)	141.4 (13.2)	3472.7 (928.4)	26.2 (0.3)	203.4 (14.6)	100	44.7 (0.4)	228.8 (15.3)	100
Brute-force	50	5	132.8 (37.0)	14736.9 (2832.4)	7122.4 (889.5)	10638.1 (3512.0)	74.6 (0.3)	5586.2 (71.7)	4	207.0 (0.4)	8685.1 (92.9)	79
kd-tree	50	5	133.4 (35.5)	14877.3 (2806.8)	7166.0 (885.6)	10687.8 (3570.3)	75.1 (1.3)	5587.3 (68.6)	4	202.6 (4.2)	8685.3 (88.5)	79
LSH (*L*: 20, *c*: 150)	50	5	76.9 (17.1)	14772.0 (2983.9)	7134.2 (950.1)	10530.9 (3681.8)	49.8 (0.3)	5590.5 (68.6)	4	107.3 (0.6)	8691.0 (89.4)	80
LSH (*L*: 5, *c*: 150)	50	5	73.8 (15.0)	15041.0 (2891.2)	7386.4 (930.1)	10949.9 (3734.6)	47.7 (0.3)	5723.0 (70.7)	4	99.5 (0.4)	8888.4 (93.0)	79
LSH (*L*: 20, *c*: 300)	50	5	76.0 (16.0)	14768.7 (2926.6)	7131.4 (927.1)	10858.1 (3707.3)	50.0 (0.4)	5587.6 (69.3)	5	104.6 (0.8)	8692.8 (87.1)	79
Brute-force	100	5	162.7 (46.3)	13887.6 (3033.0)	6279.6 (797.0)	10083.6 (3407.6)	104.9 (0.4)	5242.3 (71.8)	9	262.5 (0.6)	7748.9 (92.9)	89
kd-tree	100	5	166.5 (46.3)	13831.3 (2982.6)	6265.8 (789.7)	10262.5 (3413.3)	108.3 (0.9)	5242.9 (66.1)	10	268.9 (3.2)	7753.2 (82.3)	90
LSH (*L*: 20, *c*: 150)	100	5	111.5 (25.0)	13793.8 (3035.7)	6252.3 (806.0)	10172.0 (3399.0)	80.2 (0.3)	5246.5 (69.9)	11	162.4 (0.5)	7753.9 (87.1)	90
LSH (*L*: 5, *c*: 150)	100	5	111.6 (22.2)	14103.0 (2898.4)	6583.0 (789.0)	10415.3 (3550.8)	80.0 (0.4)	5432.5 (68.3)	8	156.2 (0.7)	8057.9 (88.3)	89
LSH (*L*: 20, *c*: 300)	100	5	113.5 (23.7)	13939.1 (2960.7)	6305.5 (776.0)	10164.7 (3335.4)	81.9 (0.4)	5258.1 (71.2)	10	161.4 (0.6)	7762.4 (90.3)	89

Table II. Average results of 1000 test runs for house environment, with standard deviations shown in parentheses. SR means success rate. L means the number of hash tables, and c means the number of clusters.

			Path found				10,000 nodes			30,000 nodes		
Method	*k*	Robots	Time (s)	Nodes	Components	Length	Time (s)	Components	SR	Time (s)	Components	SR
Brute-force	50	1	10.2 (13.6)	7917.1 (6995.0)	9.7 (3.7)	1407.1 (228.4)	10.0 (0.1)	8.1 (2.2)	63	68.8 (0.4)	3.9 (1.4)	90
kd-tree	50	1	3.7 (3.0)	8080.2 (7320.6)	9.6 (3.8)	1406.6 (237.1)	4.4 (0.1)	8.2 (2.3)	63	12.9 (0.4)	3.9 (1.4)	90
LSH (*L*: 20, *c*: 150)	50	1	4.0 (3.4)	8234.5 (7353.1)	9.4 (3.7)	1397.7 (213.9)	4.6 (0.1)	8.3 (2.3)	61	15.3 (0.1)	3.9 (1.3)	89
LSH (*L*: 5, *c*: 150)	50	1	3.9 (3.0)	8553.2 (7716.8)	9.7 (3.7)	1405.0 (234.6)	4.4 (0.1)	8.8 (2.4)	60	12.8 (0.1)	4.3 (1.6)	89
LSH (*L*: 20, *c*: 300)	50	1	4.5 (3.4)	8507.1 (7441.7)	9.5 (3.7)	1413.3 (244.5)	5.2 (0.1)	8.3 (2.3)	60	14.8 (0.1)	4.0 (1.4)	90
Brute-force	100	1	7.4 (10.0)	5743.0 (5593.0)	9.2 (3.4)	1409.2 (233.2)	11.1 (0.2)	6.2 (1.9)	83	68.8 (0.2)	2.7 (0.8)	98
kd-tree	100	1	3.8 (2.6)	5679.9 (5389.9)	9.3 (3.4)	1392.0 (231.8)	5.8 (0.2)	6.3 (1.9)	83	15.6 (0.3)	2.6 (0.8)	98
LSH (*L*: 20, *c*: 150)	100	1	4.2 (2.7)	5617.6 (5251.5)	9.1 (3.3)	1411.9 (237.3)	6.3 (0.2)	6.2 (1.9)	82	17.2 (0.2)	2.7 (0.9)	98
LSH (*L*: 5, *c*: 150)	100	1	4.4 (2.9)	5674.2 (5640.4)	9.3 (3.4)	1405.8 (229.4)	6.9 (0.2)	6.4 (1.9)	81	15.5 (0.2)	2.9 (1.0)	98
LSH (*L*: 20, *c*: 300)	100	1	5.0 (3.4)	5686.0 (5541.2)	9.2 (3.3)	1387.6 (236.2)	8.1 (0.2)	6.2 (1.9)	82	17.8 (0.2)	2.7 (0.9)	97
Brute-force	50	2	46.4 (35.6)	13486.9 (7586.5)	140.9 (11.2)	4580.3 (1121.5)	26.6 (0.3)	138.1 (9.8)	34	141.5 (0.6)	147.2 (10.5)	88
kd-tree	50	2	26.9 (14.4)	13405.1 (7285.8)	140.9 (10.7)	4571.8 (1100.0)	20.1 (0.4)	137.9 (10.2)	33	60.2 (1.6)	147.2 (10.5)	87
LSH (*L*: 20, *c*: 150)	50	2	20.9 (11.0)	13142.5 (7388.6)	140.8 (10.6)	4692.3 (1172.6)	16.2 (0.2)	138.2 (10.4)	36	47.1 (0.4)	146.9 (10.2)	87
LSH (*L*: 5, *c*: 150)	50	2	20.5 (9.1)	14092.3 (7349.5)	149.6 (11.0)	4677.7 (1158.2)	15.7 (0.2)	148.6 (10.9)	30	40.3 (0.3)	154.9 (10.6)	87
LSH (*L*: 20, *c*: 300)	50	2	20.7 (9.7)	13124.8 (7136.1)	140.5 (10.9)	4667.0 (1140.7)	16.6 (0.2)	137.6 (10.1)	35	43.8 (0.4)	146.5 (10.5)	86
Brute-force	100	2	37.9 (29.7)	10517.4 (6565.2)	128.0 (11.8)	4575.1 (1124.9)	31.8 (0.6)	124.1 (9.2)	53	149.1 (1.1)	129.4 (9.2)	97
kd-tree	100	2	29.1 (17.1)	10547.3 (6490.6)	128.0 (11.6)	4601.9 (1111.0)	27.6 (0.8)	123.7 (9.4)	53	80.7 (2.3)	129.3 (9.2)	97
LSH (*L*: 20, *c*: 150)	100	2	22.5 (11.3)	10639.8 (6671.6)	127.8 (11.7)	4643.1 (1121.3)	21.9 (0.6)	123.5 (9.3)	52	54.7 (1.0)	129.4 (10.0)	96
LSH (*L*: 5, *c*: 150)	100	2	23.7 (10.4)	11348.8 (6840.5)	135.1 (10.4)	4598.6 (1147.9)	22.6 (0.7)	134.3 (9.9)	47	49.6 (1.1)	138.9 (10.2)	94
LSH (*L*: 20, *c*: 300)	100	2	23.6 (10.7)	10755.7 (6667.4)	127.6 (11.7)	4609.7 (1128.5)	23.3 (0.6)	123.4 (9.7)	52	52.3 (1.0)	129.0 (10.0)	96
Brute-force	50	3	188.9 (52.9)	23718.1 (4983.7)	2488.3 (130.8)	9969.3 (2393.7)	57.1 (0.3)	2069.3 (40.0)	0	263.5 (0.6)	2604.7 (45.6)	6
kd-tree	50	3	165.3 (40.2)	23519.5 (4618.2)	2486.3 (116.1)	9968.1 (2462.5)	56.3 (0.6)	2070.6 (38.5)	0	224.9 (3.7)	2607.8 (44.8)	7
LSH (*L*: 20, *c*: 150)	50	3	102.9 (19.9)	23839.2 (4426.2)	2510.0 (96.7)	10260.0 (2340.1)	42.5 (0.3)	2072.6 (39.0)	0	130.9 (0.6)	2605.6 (47.6)	6
LSH (*L*: 5, *c*: 150)	50	3	94.9 (20.5)	23999.6 (5369.6)	2702.9 (147.8)	11066.4 (3196.6)	41.0 (0.3)	2253.1 (42.0)	0	117.7 (0.6)	2799.6 (50.3)	4
LSH (*L*: 20, *c*: 300)	50	3	98.2 (17.7)	23854.8 (4389.1)	2504.8 (113.4)	9951.1 (2608.4)	42.2 (0.3)	2075.6 (40.6)	0	122.8 (0.7)	2610.0 (46.9)	7
Brute-force	100	3	209.7 (57.7)	23398.6 (4985.5)	2238.0 (101.9)	9641.4 (2746.8)	70.2 (0.4)	1902.4 (37.3)	0	292.4 (1.2)	2331.1 (42.9)	16
kd-tree	100	3	201.1 (51.7)	23293.6 (4885.1)	2240.7 (98.9)	9637.0 (2571.8)	71.6 (0.7)	1902.4 (37.6)	0	275.0 (3.7)	2330.0 (43.8)	16
LSH (*L*: 20, *c*: 150)	100	3	125.8 (25.9)	23290.2 (4950.2)	2240.5 (121.2)	9908.8 (2670.1)	56.2 (0.5)	1907.6 (37.2)	0	161.2 (1.6)	2332.5 (43.2)	15
LSH (*L*: 5, *c*: 150)	100	3	123.0 (22.2)	24118.9 (4804.3)	2468.8 (98.6)	10139.9 (2508.8)	56.1 (0.5)	2091.9 (41.5)	0	149.6 (1.5)	2551.9 (44.5)	9
LSH (*L*: 20, *c*: 300)	100	3	121.3 (20.8)	23424.8 (4409.4)	2250.4 (79.3)	9852.7 (2426.8)	56.3 (0.4)	1912.9 (37.9)	0	152.1 (1.3)	2335.7 (43.0)	14

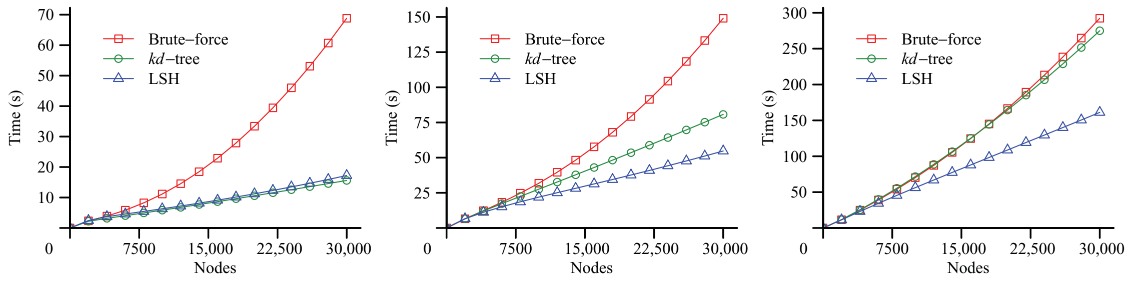
Table III. Average results of 1000 test runs for wall environment, with standard deviations shown in parentheses. SR means success rate. L means the number of hash tables, and c means the number of clusters.

			Path found				10,000 nodes			20,000 nodes		
Method	*k*	Robots	Time (s)	Nodes	Components	Length	Time (s)	Components	SR	Time (s)	Components	SR
Brute-force	50	3	0.7 (0.5)	1008.7 (557.6)	2.7 (1.3)	1744.0 (539.2)	18.6 (0.1)	9.3 (3.0)	100	71.4 (0.2)	17.6 (4.1)	100
kd-tree	50	3	0.8 (0.5)	1044.2 (587.3)	2.7 (1.3)	1748.7 (525.8)	19.2 (0.4)	9.3 (2.9)	100	67.3 (1.0)	17.5 (4.1)	100
LSH (*L*: 20, *c*: 150)	50	3	0.6 (0.3)	1066.4 (615.6)	2.8 (1.3)	1786.2 (552.0)	4.2 (0.1)	9.4 (2.9)	100	12.5 (0.2)	17.7 (4.2)	100
LSH (*L*: 5, *c*: 150)	50	3	0.7 (0.4)	1052.9 (616.9)	2.8 (1.3)	1730.3 (517.6)	2.9 (0.1)	9.3 (2.9)	100	6.6 (0.2)	17.6 (4.1)	100
LSH (*L*: 20, *c*: 300)	50	3	0.8 (0.4)	1063.2 (647.4)	2.9 (1.3)	1758.2 (534.0)	4.1 (0.1)	9.4 (2.9)	100	10.0 (0.2)	17.6 (4.1)	100
Brute-force	100	3	0.7 (0.3)	606.3 (287.6)	2.4 (1.2)	1689.8 (502.1)	19.1 (0.2)	9.4 (2.9)	100	72.4 (0.2)	17.7 (4.1)	100
kd-tree	100	3	0.7 (0.3)	596.0 (271.9)	2.4 (1.1)	1676.7 (479.5)	22.3 (0.4)	9.4 (2.8)	100	80.9 (1.1)	17.8 (4.1)	100
LSH (*L*: 20, *c*: 150)	100	3	0.7 (0.3)	590.7 (270.6)	2.4 (1.2)	1704.2 (533.5)	4.6 (0.2)	9.4 (2.9)	100	13.0 (0.2)	17.8 (4.1)	100
LSH (*L*: 5, *c*: 150)	100	3	0.7 (0.3)	599.9 (273.6)	2.5 (1.2)	1709.0 (541.1)	4.6 (0.2)	9.3 (2.9)	100	8.5 (0.3)	17.6 (4.1)	100
LSH (*L*: 20, *c*: 300)	100	3	0.7 (0.3)	599.8 (281.7)	2.4 (1.2)	1681.1 (517.7)	5.2 (0.2)	9.2 (2.9)	100	11.2 (0.2)	17.5 (4.0)	100
Brute-force	50	5	12.8 (7.0)	4901.0 (1715.4)	11.9 (3.7)	5426.9 (2047.6)	35.6 (0.5)	13.4 (3.5)	98	123.2 (0.6)	25.2 (4.9)	100
kd-tree	50	5	15.2 (9.7)	4919.4 (1749.2)	11.8 (3.7)	5656.3 (2068.0)	46.3 (0.8)	13.5 (3.8)	98	175.6 (1.3)	25.2 (5.1)	100
LSH (*L*: 20, *c*: 150)	50	5	7.0 (2.4)	4978.8 (1783.5)	11.9 (3.7)	5540.5 (1945.6)	13.1 (0.6)	13.4 (3.5)	98	32.7 (0.8)	25.1 (4.8)	100
LSH (*L*: 5, *c*: 150)	50	5	7.0 (1.8)	5453.7 (1967.6)	15.2 (4.4)	5742.0 (2035.8)	10.2 (0.6)	15.4 (3.8)	97	18.8 (0.6)	26.5 (5.1)	100
LSH (*L*: 20, *c*: 300)	50	5	7.4 (2.2)	5097.2 (1779.0)	12.0 (3.8)	5634.1 (2149.7)	12.4 (0.6)	13.4 (3.4)	98	26.2 (0.6)	25.2 (4.8)	100
Brute-force	100	5	8.8 (3.4)	3085.3 (1035.4)	9.5 (3.7)	5195.7 (1981.0)	37.3 (0.8)	12.8 (3.3)	100	125.3 (0.8)	24.4 (4.8)	100
kd-tree	100	5	9.9 (4.2)	3160.8 (1059.4)	9.3 (3.6)	5295.4 (1909.1)	48.1 (1.0)	12.9 (3.5)	100	177.8 (1.5)	24.6 (4.8)	100
LSH (*L*: 20, *c*: 150)	100	5	7.0 (1.8)	3234.8 (1161.9)	9.5 (3.7)	5315.4 (1944.5)	15.0 (0.8)	12.9 (3.5)	100	34.6 (0.9)	24.6 (4.9)	100
LSH (*L*: 5, *c*: 150)	100	5	8.0 (1.9)	3263.7 (1193.4)	10.1 (3.6)	5298.3 (1960.5)	13.2 (0.9)	13.7 (3.5)	100	21.9 (0.9)	25.1 (4.9)	100
LSH (*L*: 20, *c*: 300)	100	5	7.9 (1.8)	3297.4 (1136.4)	9.5 (3.7)	5347.8 (1967.1)	14.7 (0.8)	12.8 (3.5)	100	28.4 (0.8)	24.4 (5.0)	100
Brute-force	50	7	158.7 (26.8)	17681.7 (1827.5)	129.4 (12.4)	11655.2 (4808.5)	60.4 (0.2)	165.2 (11.1)	0	195.5 (0.6)	126.4 (12.1)	15
kd-tree	50	7	204.6 (37.4)	17722.0 (1957.7)	128.7 (12.6)	11757.0 (4587.6)	72.7 (0.5)	165.0 (11.4)	0	253.3 (1.0)	126.4 (12.2)	15
LSH (*L*: 20, *c*: 150)	50	7	63.4 (8.0)	17684.2 (1694.8)	128.8 (12.4)	11318.4 (4435.1)	30.4 (0.2)	164.9 (11.0)	0	74.7 (0.9)	125.8 (11.9)	14
LSH (*L*: 5, *c*: 150)	50	7	47.5 (4.8)	18193.5 (1704.4)	247.9 (24.0)	11782.1 (4904.7)	25.1 (0.1)	310.1 (20.2)	0	52.7 (0.4)	244.9 (19.7)	5
LSH (*L*: 20, *c*: 300)	50	7	57.1 (5.8)	18014.0 (1572.0)	127.2 (10.1)	11539.6 (4034.4)	29.1 (0.2)	166.3 (11.3)	0	64.7 (0.7)	125.9 (11.5)	12
Brute-force	100	7	140.0 (35.5)	15157.2 (2552.0)	58.8 (10.6)	13131.6 (5109.7)	73.5 (0.3)	85.3 (9.0)	2	212.1 (2.2)	50.7 (6.9)	72
kd-tree	100	7	174.6 (45.5)	15253.5 (2462.0)	58.6 (10.6)	13055.3 (5029.1)	86.2 (0.4)	85.3 (9.0)	2	270.4 (2.3)	50.6 (7.2)	72
LSH (*L*: 20, *c*: 150)	100	7	70.1 (12.3)	15340.6 (2479.1)	59.6 (10.7)	12939.8 (5193.8)	43.7 (0.3)	87.5 (8.9)	1	91.7 (2.3)	51.3 (7.1)	72
LSH (*L*: 5, *c*: 150)	100	7	65.1 (7.0)	16876.3 (1975.0)	111.0 (15.8)	13700.1 (5004.6)	39.9 (0.2)	163.6 (13.0)	0	74.9 (2.0)	99.8 (11.8)	51
LSH (*L*: 20, *c*: 300)	100	7	65.1 (9.4)	15445.5 (2322.8)	60.8 (10.8)	13231.0 (4862.5)	42.5 (0.4)	89.8 (9.1)	0	81.7 (2.3)	52.1 (7.4)	66



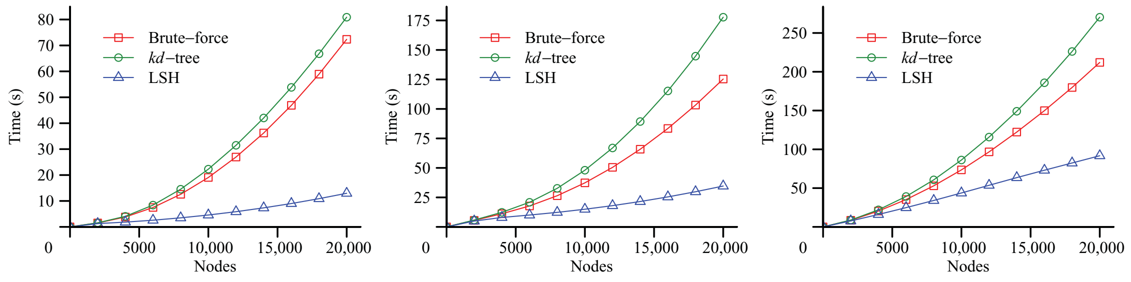
(a) 1 robot body (b) 3 robot bodies (c) 5 robot bodies

Fig. 5. Graphs show how much time was used to build the roadmap in asteroids environment with different methods.



(a) 1 robot body (b) 2 robot bodies (c) 3 robot bodies

Fig. 6. Graphs show how much time was used to build the roadmap in house environment with different methods.



(a) 3 robot bodies (b) 5 robot bodies (c) 7 robot bodies

Fig. 7. Graphs show how much time was used to build the roadmap in wall environment with different methods.

dimensions for a configuration space. By decreasing the number of hash tables to five, the planner worked faster, but the quality of the roadmap decreased. Increasing the number of centroids to 300 did not have as large effects, but, for example, the success rates in the wall environment decreased, as can be seen in Table III. However, it seems that there is no need for a major fine tuning of the parameters to get good results with the LSH method.

By comparing the time required to build the whole roadmap with different values for k, it can be seen that increasing the value also increases the used time. On the other hand, when k is large, the success rates are higher and the predefined query can be solved with a smaller roadmap. This is because k determines the number of configurations to which each new configuration is tried to be connected with a local planner. The planner requires more computation time with large values, but at the same time the connectivity of the roadmap increases. When the connectivity is high, the predetermined query can be solved with a smaller number of nodes and with a better success rate than with lower connectivity.

It can be noted from the results that the time required to build the roadmap grows also when the number of robot bodies increases, no matter what the nearest neighbor method is used. There are several reasons for this. The first reason is that the collisions must be checked for each robot body, which obviously requires more time as the number of bodies grows. The second reason is that when the robot consists of multiple bodies, the robot can collide with itself in some configurations. Therefore, it is not enough to check collisions with static obstacles because also self-collisions must be checked. The third reason has to do with a local planner which tries to connect a new configuration with only those configurations that do not belong to the same component (see line 8 in Algorithm 1). When the number of robot bodies grows, it will be increasingly difficult for the local planner to find a free path between two configurations. Therefore, more and more paths must be checked. This can also be seen from the number of components that grows rapidly when the number of robot bodies increases.

6. Conclusions

Probabilistic roadmap planners must use the nearest neighbor methods to retrieve a set of neighboring configurations when a new configuration is added to the roadmap. Neighbors are usually searched for by exact methods, which can unfortunately became a major bottleneck for the performance. This can occur when the roadmap size grows and especially when the configuration space is high-dimensional.

In this paper, we investigated how approximate the nearest neighbor methods work with PRM planners. We focused on LSH, which is nowadays quite a popular method for approximate neighbor search and has been successfully used in many applications. We compared the LSH method with two exact methods, which were the brute-force search and the *kd*-tree method.

Our experiments showed that it is indeed feasible to use approximate nearest neighbor search when constructing the roadmap. The approximate search can speed up the roadmap construction phase considerably. Our experiments also showed that *kd*-tree is not a good method to be used in high-dimensional spaces as it will eventually became even slower than the brute-force method.

We also compared the quality of the roadmaps produced with different methods. With good parameters for the LSH method, there were no significant differences in quality between the LSH method and the exact methods. The roadmap was able to solve the predetermined query with the same success rate, and the length of the found path was approximately the same. We also measured the number of roadmap components in certain roadmap sizes and noted that there were no big differences between the methods.

The problem with the LSH method is that it requires parameters that must be selected manually to suit each motion planning problem. However, the LSH methods generally work faster than exact algorithms and produce reasonably good quality roadmaps even with bad choices of parameter values. Because of this, it seems that there is no need for an extreme fine tuning of parameters. Still it would be worth to investigate whether these parameters could be selected automatically. Another interesting idea for the future research is to compare how other approximate the nearest neighbor methods work when compared with LSH.

Acknowledgment

The research of the first author was supported by the Tampere Doctoral Programme in Information Science and Engineering (TISE).

References

1. J.-C. Latombe, *Robot Motion Planning* (Kluwer, Boston, MA, 1991).
2. H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations* (MIT Press, Cambridge, MA, 2005).
3. S. M. LaValle, *Planning Algorithms* (Cambridge University Press, Cambridge, UK, 2006).
4. N. Dadkhah and B. Mettler, "Survey of motion planning literature in the presence of uncertainty: Considerations for UAV guidance," *J. Intell. Robot. Syst.* **65**, 233–246 (2012).
5. J. Smed and H. Hakonen, "Path Finding," **In**: *Algorithms and Networking for Computer Games* (John Wiley, 2006) pp. 97–113.

6. I. Al-Bluwi, T. Siméon and J. Cortés, "Motion planning algorithms for molecular simulations: A survey," *Comput. Sci. Rev.* **6**(4), 125–143 (2012).
7. T. Lozano-Pérez, "Spatial planning: A configuration space approach," *IEEE Trans. Comput.* **C-32**(2), 108–120 (1983).
8. J. H. Reif, "Complexity of the Mover's Problem and Generalizations," *Proceedings of the IEEE Symposium on Foundations of Computer Science*, San Juan, Puerto Rico (Oct. 29–31, 1979) pp. 421–427.
9. J. F. Canny, *The Complexity of Robot Motion Planning* (MIT Press, Cambridge, MA, 1988).
10. M. H. Overmars, "A Random Approach to Motion Planning," *Technical Report* RUU-CS-92-93, Department of Computer Science, Utrecht University, the Netherlands (1992).
11. L. Kavraki and J.-C. Latombe, "Randomized Preprocessing of Configuration Space for Fast Path Planning," *Proceedings of the IEEE International Conference on Robotics and Automation*, San Diego, CA (May 8–13, 1994) pp. 2138–2145.
12. L. E. Kavraki, P. Švestka, J.-C. Latombe and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Autom.* **12**(4), 566–580 (1996).
13. L. Jaillet and T. Siméon, "A PRM-Based Motion Planner for Dynamically Changing Environments," *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan (Sept. 28–Oct. 2, 2004) pp. 1606–1611.
14. A. Sud, R. Gayle, E. Andersen, S. Guy, M. Lin and D. Manocha, "Real-Time Navigation of Independent Agents Using Adaptive Roadmaps," *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, Newport Beach, CA (Nov. 5–7, 2007) pp. 99–106.
15. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *J. ACM* **45**(6), 891–923 (1998).
16. M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," *Proceedings of the International Conference on Computer Vision Theory and Application*, Lisboa, Portugal, (Feb. 5–8, 2009) pp. 331–340.
17. P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," *Proceedings of ACM Symposium on Theory of Computing*, Dallas, Texas, USA (May 23–26, 1998) pp. 604–613.
18. A. Gionis, P. Indyk and R. Motwani, "Similarity Search in High Dimensions via Hashing," *Proceedings of International Conference on Very Large Data Bases*, Edinburgh, Scotland (Sep. 7–10, 1999) pp. 518–529.
19. A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM* **51**(1), 117–122 (2008).
20. L. Kocis and W. J. Whiten, "Computational investigations of low-discrepancy sequences," *ACM Trans. Math. Softw.* **23**(2), 266–294 (1997).
21. N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones and D. Vallejo, "OBPRM: An Obstacle-Based PRM for 3D Workspaces," *Proceedings of Workshop on Algorithmic Foundations of Robotics*, Houston, Texas (Mar. 5–7, 1998) pp. 155–168.
22. V. Boor, M. H. Overmars and A. F. van der Stappen, "The Gaussian Sampling Strategy for Probabilistic Roadmap Planners," *Proceedings of IEEE International Conference on Robotics & Automation*, Detroit, MI (May 10–15, 1999) pp. 1018–1023.
23. D. Hsu, T. Jiang, J. Reif and Z. Sun, "The Bridge Test for Sampling Narrow Passages with Probabilistic Roadmap Planners," *Proceedings of IEEE International Conference on Robotics & Automation*, Taipei, Taiwan (Sep. 14–19, 2003) pp. 4420–4426.
24. S. A. Wilmarth, N. M. Amato and P. F. Stiller, "MAPRM: A Probabilistic Roadmap Planner with Sampling on the Medial Axis of the Free Space," *Proceedings of IEEE International Conference on Robotics & Automation*, Detroit, MI (May 10–15, 1999) pp. 1024–1031.
25. C. Holleman and L. E. Kavraki, "A Framework for Using the Workspace Medial Axis in PRM Planners," *Proceedings of IEEE International Conference on Robotics & Automation*, San Francisco, CA (Apr. 24–28, 2000) pp. 1408–1413.
26. M. Morales, L. Tapia, R. Pearce, S. Rodriguez and N. M. Amato, "A Machine Learning Approach for Feature-Sensitive Motion Planning," **In**: *Algorithmic Foundations of Robotics VI* (M. Erdmann, M. Overmars, D. Hsu and F. van der Stappen, eds.) (Springer, Berlin, 2005), pp. 361–376.
27. S. Rodriguez, S. Thomas, R. Pearce and N. M. Amato, "RESAMPL: A Region-Sensitive Adaptive Motion Planner," **In**: *Algorithmic Foundation of Robotics VII* (S. Akella, N. M. Amato, W. H. Huang and B. Mishra, eds.) (Springer, Berlin, 2008) pp. 285–300.
28. M. T. Rantanen, "A connectivity-based method for enhancing sampling in probabilistic roadmap planners," *J. Intell. Robot. Syst.* **64**(2), 161–178 (2011).
29. N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones and D. Vallejo, "Choosing good distance metrics and local planners for probabilistic roadmap methods," *IEEE Trans. Robot. Autom.* **16**(4), 442–447 (2000).
30. R. Geraerts and M. H. Overmars, "Reachability-based analysis for probabilistic roadmap planners," *Robot. Auton. Syst.* **55**(11), 824–836 (2007).
31. J. J. Kuffner, "Effective Sampling and Distance Metrics for 3D Rigid Body Path Planning," *Proceedings of IEEE International Conference on Robotics & Automation*, New Orleans, LA (Apr. 26–May 1, 2004) pp. 3993–3998.
32. R. Weber, H.-J. Schek and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proceedings of International Conference on Very Large Databases*, New York City, New York, USA (Aug. 24–27, 1998) pp. 194–205.

33. J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM* **18**(9), 509–517 (1975).
34. M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed. (Springer-Verlag, Berlin, Germany, 2000).
35. A. Yershova and S. M. LaValle, "Improving motion-planning algorithms by efficient nearest-neighbor searching," *IEEE Trans. Robot.* **23**(1), 151–157 (2007).
36. M. Ryynänen and A. Klapuri, "Query by Humming of MIDI and Audio Using Locality Sensitive Hashing," *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, Las Vegas, NV (Mar. 31–Apr. 4, 2008) pp. 2249–2252.
37. R. Buaba, A. Homaifar, M. Gebril and E. Kihn, "Satellite Image Retrieval Application Using Locality Sensitive Hashing in L_2-Space," *Proceedings of IEEE Aerospace Conference*, Big Sky, MT, USA (Mar. 5–12, 2011) pp. 1–7.
38. M. Datar, N. Immorlica, P. Indyk and V. S. Mirrokni, "Locality-Sensitive Hashing Scheme Based on p-Stable Distributions," *Proceedings of Symposium on Computational Geometry*, Brooklyn, New York, USA (Jun. 8–11, 2004) pp. 253–262.
39. L. Paulevé, H. Jégou and L. Amsaleg, "Locality sensitive hashing: A comparison of hash function types and querying mechanisms," *Pattern Recognit. Lett.* **31**(11), 1348–1358 (2010).
40. F. Aurenhammer, "Voronoi diagrams–a survey of a fundamental geometric data structure," *ACM Comput. Surv.* **23**(3), 345–405 (1991).
41. E. Larsen, S. Gottschalk, M. C. Lin and D. Manocha, "Fast Proximity Queries with Swept Sphere Volumes," *Technical Report* TR99-018, Department of Computer Science, University of North Carolina, Wilmington, NC (1999).