Constant Time Collision-Free Path Computation on Reconfigurable Mesh

Hatem M. El-Boghdadi
Computer Engineering Department, Faculty of Engineering,
Cairo University, Giza, Egypt
helboghdadi@eng.cu.edu.eg

Abstract - The reconfigurable mesh (R-Mesh) was shown to be a very powerful model capable of extremely fast solutions to many problems. R-Mesh has a wide range of applications such as arithmetic problems, image processing and robotics. The 2D R-Mesh was shown to be able to solve the path planning problem very fast.

In this paper, we propose an algorithm to compute a collision-free path between a source and a destination in an environment with the existence of obstacles. Independent of the number of obstacles, k, the proposed algorithm runs in constant time and requires $O(\log^2 N)$ pre-processing time where $N$ is the size of the R-Mesh. This is in contrast to the previous work that requires $O(k)$ time with the same pre-processing time. We then make a modification to the obtained path to enhance its length. This enhancement also requires constant time.

Keywords-parallel algorithms, reconfigurable mesh, path planning

1 Introduction

Reconfiguration is a very powerful computing paradigm, capable of extremely fast solutions to many problems. Models such as the reconfigurable mesh (R-Mesh) [9] (shown in Figure 1) have the ability to change the interconnection between processors at every step of the computation to allow efficient communication as well as performing computation faster than conventional non-reconfigurable models. A 2D R-Mesh is an array of processors with fixed external connections between each two neighboring processors. Also it has dynamically reconfigurable internal connections within each processor. This allows altering the interconnection among processors very fast, possibly at each step of the computation.

Robot motion planning algorithms aim to plan a path for a moving robot from a source location to a target location. The environment that the robot navigates through may have obstacles. The main target for the robot is to maneuver without colliding with obstacles if any. In general, the existent obstacles may be static or dynamic obstacles; i.e. moving obstacles. Restrictions on the robot movement from source to destination may take into account a number of factors. These include the robot shape and type of movement; for example translational or rotational. To simply handle the path planning problem, the image of robot and obstacles are digitized and stored in the R-Mesh, with one processor holding one pixel of the image. The R-Mesh was shown to handle these kinds of problems very fast. Also some techniques use first an algorithm to generate the configuration space. The configuration space is a slightly different image for the robot and the obstacles than the original image. It can be obtained by expanding the obstacles based on the shape of the robot and its movement. This way, the robot that is not point-like can be converted to a point-like robot. This greatly simplifies the design and analysis of the algorithm.

Since path planning problem is computationally intensive, many parallel algorithms have been proposed using various algorithmic models and assumed different robot shapes. These models include R-Mesh, hypercube computers, etc.

Tzionas et al. assumed a diamond-shaped robot and presented a parallel algorithm for collision-free path planning [12]. Jenq and Li [4] used hypercube to compute the configuration space optimally. The algorithm was shown to be optimal where it requires $O(\log n)$ time for an $n \times n$ image by using $n \times n$ Mesh of processors. D. Wang [14] proposed efficient algorithms for solving the reachability problem in one dimensional space. Dehne et al. [3] presented a systolic algorithm for computing the configuration space for obstacles in a plane for a rectilinear convex robot. The algorithm takes $O(n)$ time for an $n \times n$ image on an $n \times n$ mesh of processors.

H.C. Lee [7] studied the maze-routing problem and it was shown that the R-Mesh is suitable for developing efficient and fast algorithms to solve the maze-routing problem. The maze-routing problem has its application in the path planning problem.
the algorithm finds the tangent lines to that obstacle and finds if obstacle is found or using the combined obstacle, the combines these obstacles into one big obstacle. If only one obstacle is found, then the algorithm finds the obstacles that intersect with that line if any. Then an enhancement to the generated path is proposed. This enhancement tends to decrease the length of the generated path.

In this paper, we consider the R-Mesh for computing a collision-free path between a source and a destination in the presence of obstacles. Independent of the number of obstacles, \( k \), the algorithm runs in constant time and requires \( O(k) \) pre-processing time where \( N \) is the size of the R-Mesh. This is in contrast to the work of Wang [15] that requires \( O(k) \) time with the same pre-processing time. The main idea is to start with a straight line between the source \( s \) and the destination \( d \). We denote this line segment \( sd \). The path is generated such that it follows the straight line segment \( sd \) with going around each obstacle that intersects with this path. The algorithm generates \( 2^m \) possible paths where \( m \) is the number of obstacles that intersects with \( sd \), \( 1 \leq m \leq k \). Then an enhancement to the generated path is proposed. This enhancement tends to decrease the length of the generated path.

The next section presents some preliminaries and definitions. Section 3 describes the preprocessing operations that are applied before running the proposed algorithm. In Section 4, we describe the proposed algorithm and its time analysis. Section 5 presents an enhancement to the algorithm that could reduce the length of the generated path. In Section 6, we summarize our results and make some concluding remarks.

2 Preliminaries

In this section we introduce some preliminaries and definitions that are used throughout this paper.

2.1 Reconfigurable Mesh

An \( R \times C \) reconfigurable mesh or R-Mesh [13] consists of an \( R \)-row, \( C \)-column array of processors connected by an underlying mesh (see Figure 1). Number the rows (resp., columns) 0, 1, ..., \( R-1 \) (resp., 0, 1, ..., \( C-1 \)) as shown in Figure 1. Each processor has four ports (called North, South, East, and West ports in the obvious manner, and abbreviated \( N, S, E, \) and \( W \) (see Figure 1(b)).

Each processor can independently partition its ports to connect certain ports together leaving other ports unconnected. For example, in Figure 1(a) the top left processor connects its \( N \) port to its \( S \) port, and its \( E \) port to its \( W \) port. The corresponding partition is denoted by \( \{NS, EW\} \).
robot shape and size [6]. For example if the robot is of polygon shape, the configuration space can be computed using the R-Mesh in O(1) time. In this paper we deal with the configuration space directly; i.e. we assume it has been already computed. Figure 3 shows the expanded obstacles for a rectangular robot.

3 Pre-processing Operations

The preprocessing operation, convexaziation, handles all the obstacles in the environment. The process builds a convex hull for each obstacle and enumerate the extreme points of each convex. Figure 4 shows an obstacle after applying the operation of convexaziation. The target is to identify the extreme points of the convex hull of the given polygon \( H \). The figure shows the polygon after enumerating its extreme points. The work of Miller et al. [9] performs this operation on all the obstacles in the environment in \( O(\log^2 N) \) time where \( N \) is the size of the R-Mesh. After enumerating the extreme points, each extreme point processor has a flag to identify it as extreme point. Each extreme point processor has the following information (1) The ID number of the polygon this extreme point belongs to. (2) The number of the extreme point in the polygon. (3) The locations of the previous and the following extreme points. In other words, each extreme point contains segments information associated with it.

We assume that after convexization, \( s \) and \( d \) are not covered by any convex polygon. If that is not the case, a constant time operation treatment discussed later in section 6 reduces the problem to the assumed situation.

4 Constant Time Algorithm

In this section we describe a constant time algorithm to solve the path planning problem on the R-Mesh. In section 5, we propose a constant time operation that enhances the generated path. Given an environment in the presence of obstacles, it is required to find a collision-free path from a source location \( s \), to a destination location \( d \). We assume that the obstacles to be disjoint convex or concave polygons. If two polygon images intersect, we consider them one polygon. The input to the R-Mesh is an \( n \times n \) image that represents the environment. The image is digitized, stored to the R-MESH, one pixel per processor. There is a flag associated with each processor. The flag has a value of 1 or 0 based on the digitized image. We use an \( n \times n \) R-Mesh to solve the path planning problem.

Generally speaking, the idea is to start with a straight line between the source \( s \) and the destination \( d \). The final path tries to follow the line segment \( \overline{sd} \). If \( \overline{sd} \) does not intersect with any obstacle, then \( \overline{sd} \) represents the final path. If \( \overline{sd} \) intersects with a number of obstacles, then the final path follows the straight line segment \( \overline{sd} \) with going around each

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{Expanding the original obstacles and reducing the robot to a point \( s \)}
\end{figure}
obstacle that intersects with this path. We show that this path can be computed in constant time. Since going around an obstacle could be in clockwise or counter clockwise direction, then there are $2^m$ possible paths from $s$ to $d$ where $m$ is the number of obstacles that intersects with segment $sd$, $1 \leq m \leq k$.

Figure 5 shows the same environment as the presented in Figure 2. Figure 5 shows the path from the $s$ to the destination $d$ provided by the proposed algorithm.

4.1 Basic Operations

In this section we develop some operations on the R-Mesh that will be used in the proposed algorithm in Section 4.2.

- Broadcasting Data
  Once the buses of R-Mesh are constructed, data movement on the R-Mesh requires constant time. The information written on the bus by any processor incident on the bus reaches all the processors incident on the bus in constant time. To broadcast data to all processors of the R-Mesh, partition all processors as $NSEW$. If any processor write information on the any port, it reaches all processors in $O(1)$ time. Thus broadcasting data to all processors requires constant time.

- Adding a new point as an extreme point
  Let $H$ be the convex hull of an obstacle $O$. Assume that the extreme points are enumerated in counter clockwise direction. Each extreme point knows its ID and the locations of preceding and following extreme points. In other words, each extreme point contains segments information associated with it. Consider an edge $e$ with two extreme points $x$ and $y$ enumerated as $i$ and $i+1$. The target is to add a new point $z$ that is located on the segment as an extreme point. The operation uses the local bus via convex hull $H$. Let the processor representing $z$ cut the local bus from $x$ to $y$. The processor $z$ exchanges the data with extreme points $x$ and $y$ using the local bus. The data exchanged include IDs of extreme points and their locations. Now processor $x$ sets its following extreme point as $z$ and processor $y$ sets its preceding extreme point as $z$. Processor $z$ sets its preceding and following extreme points as $x$ and $y$ respectively while the processors $x$ and $y$ keep their IDs. The processor $z$ sets its ID to a special ID to indicate that $z$ is an added extreme point. The described operations require constant time. Figure 6 shows the operation of adding a new extreme point to a polygon.

- Constructing a bus from $s$ to $d$
  The target is to construct a bus from $s$ to $d$ on the R-Mesh that represents the segment $sd$ as follows. Given that the locations $s$ and $d$ are known to all processors in the R-Mesh, each processor determines whether or not it belongs to the segment $sd$. Then each processor that belongs to the segment $sd$ computes the position of its preceding processor and the position of its following processor on the path from $s$ and $d$. This allows each processor to configure its ports to connect its preceding processor and its following processor. For example, if a processor $p$ finds that its preceding (resp. following) processor is the processor on its West (resp. North), then processor $p$ configure its ports $\{NW, E, S\}$ to connect the preceding and following processors. Constructing the bus includes a constant number of operations which can be done in constant time.
4.2 The Proposed Algorithm

In this section, we describe the proposed constant-time algorithm. We assume an \( n \times n \) R-Mesh as the working environment. Again, the image of the obstacles is digitized, input and stored one pixel per processor. There are \( k \) disjoint polygonal obstacles. Also, we assume that the robot has been reduced to a point using the constant time algorithm proposed in [6]. Given the source point \( s \) and the destination point \( d \), it is required to compute a collision-free path between \( s \) and \( d \). Let \( P_s \) and \( P_d \) be the two processors that represent the positions of \( s \) and \( d \) respectively. The high level description of the proposed algorithm is as follows.

Collision-Free Path Algorithm

Input: The image of the obstacles, the positions of source \( s \) and destination \( d \).

Output: A collision-free path between \( s \) and \( d \).

Model: An \( N \)-processor, 2D-EREW R-Mesh

Step 1. Processors \( P_s \) and \( P_d \) broadcast their positions to all processors of the R-Mesh.

Step 2. All edges of all convex hulls calculate whether they intersect with the segment \( sd \) or not and determine the points of intersections if any. Let convex hull \( H_i \) intersects with \( sd \) at points \( l \) (l-point) and \( r \) (r-point) where \( i \in \{1, .., k\} \). Let \( S = \{H_i \mid i \in 1, k\} \) be the set of convex hulls that intersect with \( sd \) where \( |S| = m \).

Step 3. For all \( H_i \in S \), add points \( l \) and \( r \) to the extreme points of convex hull \( H_i \).

Step 4. Construct a bus from \( s \) to \( d \), default bus, and let all processors \( l \) and \( r \) cut the bus. Now, the bus from \( s \) to \( d \) consists of a number of segments. The first segment is from \( s \) to an \( l \)-point and the last segment is from \( r \)-point to \( d \).

Step 5. Each two neighbouring processors cutting the default bus exchange their positions.

Step 6. Determine the collision-free path from \( s \) to \( d \) as a number of consecutive segments as follows.

- First segment is from \( s \) to the following \( l \)-point along \( sd \).
- The segment from any \( r \)-point processor to the following \( l \)-point processor is along \( sd \).
- The segment from any \( l \)-point processor to the following \( r \)-point processor is composed of a number of sub-segments along the convex hull of the obstacle and taken from the enumeration in the pre-processing.
- Last segment is from an \( r \)-point to \( d \) along \( sd \).

Now we describe the algorithm in detail. Step 1 broadcasts the positions of the source point and the destination point to all the processors on the R-Mesh. This enables all processors to compute the line segment \( sd \). In Step 2, each edge belongs to a polygon determine whether or not it intersects with segment \( sd \). If an edge \( e \) intersects with segment \( sd \), then the terminal points of the edge \( e \) determine the intersection point. Note that if a polygon \( H_i \) intersects with segment \( sd \), then it intersects in two points \( l \) (l-point) and \( r \) (r-point). The \( l \)-point (resp. \( r \)-point) is the one that is on the side of the source (resp. destination). Figure 7 shows four intersection points for the obstacles \( O_1 \) and \( O_2 \) with \( sd \). Let the set \( W \) contains all the intersection points; i.e. \( W = \{l_i, r_i \mid i \in 1, k\} \). Step 3 adds the two points \( l_i \) and \( r_i \) to the extreme points of the polygon \( H_i \), \( i \in 1, k \). Thus, each point \( l_i \) knows its preceding and following extreme point in the convex hull of the obstacle, same for point \( r_i \). Since there are two paths around the obstacle to reach \( r_i \) from \( l_i \), then a certain rotation direction should be decided. Here we assume, without loss of generality, that we go around the obstacle in clockwise direction. Note that the counter clockwise direction could be also followed. In Step 4, the algorithm constructs a bus, default bus, on the R-Mesh from \( s \) to \( d \), as shown in Section 4.1 and let all processors \( l_i \) and \( r_i \), \( i \in 1, k \), cut the bus. In other words, the constructed bus now consists of a number of segments. There is a processor belongs to the set \( W \) between each two consecutive segments. In Step 5, each two neighbouring processors in \( W \) exchange their positions. This enables the start point, \( s \), to know the following extreme point (an \( l \)-point) and also enables the last intersection point (an \( r \)-point) to know that the next point is the destination, \( d \). Now the point \( s \) knows the first intersection extreme point (an \( l \)-point) and the last intersection extreme point (an \( r \)-point) knows the following point, \( d \). Also along the path from \( s \) to \( d \), each extreme point knows the following extreme point. Step 6 determines the final collision-free path from \( s \) to \( d \). The path is composed of a number of segments. All the points \( l_i \) and \( r_i \), \( i \in 1, k \), belong to the final path and each represents a turning point on the path from \( s \) to \( d \).

The first segment is from \( s \) to the following \( l \)-point and follows the segment \( sd \). The last segment is from an \( r \)-point to \( d \) and follows the segment \( sd \). A segment from any \( r \)-point processor to the following \( l \)-point processor follows the segment \( sd \). A segment from any \( l \)-point processor to the following \( r \)-point processor follows the convex hull of the obstacle and is taken from the enumeration in the pre-processing in Section 3.
4.3 Time Analysis

In this section, we show that the algorithm runs in constant time. Step 1 broadcasts the two positions for $s$ and $d$ to all processors of the R-Mesh. Broadcasting on the R-Mesh requires constant time as shown in Section 3. Step 2 calculates the intersection points for every edge with segment $sd$. Since each extreme point in a convex hull contains the information of the two segments on which it is incident. Therefore, all the intersection points can be computed in parallel in $O(1)$ time. Step 3 uses the sub bus within the polygon to add $l_i$ and $r_i$ to the extreme points of the polygon as shown before in $O(1)$ time. In step 4, all processors use the positions of $s$ and $d$ to configure their ports in parallel such that the segment $sd$ is constructed. The computation and configuration is done in constant time. Step 5 sends information on the constructed segments between each two neighbouring processors from Step 2. Sending constant number of variables on the bus requires $O(1)$ time. Step 6 requires local computation at all points $l_i$ and $r_i$, $1 \leq i \leq h$, $h \leq k$ and this requires $O(1)$ time. Thus, the whole algorithm runs in constant time.

5 Enhancing the path length

In this section, we present an enhancement to the proposed algorithm that could make a modification to the generated path. The target is to decrease the length of the generated path using constant time operations. The operations try to replace a segment of the path by another segment if this would shorten the path length. In particular, the operation tries to replace the segment $l_{(i)}y_i$ of the path, by another segment $l_{(s(i))}y_i$ if the segment of $l_{(s(i))}y_i$ is outside obstacle $O_i$. Point $l_{(i)}$ is the point of intersection between the segment $sd$ and the perpendicular line to $sd$ that passes through $y_i$. Figure 8 shows an example of the segment $sd$ that intersects with two obstacles $O_1$ and $O_2$. The path generated by the algorithm is shown in bold. However, when applying the operation to the segment $l_{(i)}y_i$ of the path, this segment is replaced by another segment $l_{(s(i))}y_i$ that would make the path shorter. The same operation is applied to the segment $w_1y_1$ and is replaced by the segment $w_1r_{s(i)}$ because it happened to be of shorter length. If the new segment lies within the obstacle itself, then no replacement is done. This is the case for $O_2$ in Figure 8. The figure shows the path in bold when applying this operation to the path of Figure 5.

The above operation of replacing one segment by another can be done in constant time. Point $y_i$ (respectively $w_i$) computes the point $l_{(i)}$ (respectively $r_{(i)}$) in constant time. If the computed point lies within the obstacle $O_i$ then no replacement is done. If the computed point, for example $l_{(i)}$, lies outside the obstacle $O_i$ then the replacement would make the path shorter. In this case, $y_i$ informs $l_i$ and $l_{(i)}$ by this replacement over the local buses in constant time.
6 Source and Destination within convex hulls

In section 4 we proposed the algorithm assuming that the source and destination points, \( s \) and \( d \), are outside any convex hull. In other words, after preprocessing phase neither \( s \) nor \( d \) is located within any convex hull of an obstacle. In this section, we deal with the case that \( s \) or \( d \) or both are within a convex hull. Figure 9 shows an example where the source \( s \) is within convex hull of obstacle \( O_i \). In such case, a minor change to the algorithm handles this situation. In Step 2 of the algorithm, after determining the intersection points, the source \( s \) and the first intersection point exchange information about their status. The status of a point could be an \( l \)-point (resp. \( r \)-point). If the source \( s \) found that the next point is an \( l \)-point, then the algorithm continues as described in Section 4. The \( l \)-point reaches the \( r \)-point through the convex hull of the obstacle. If the source \( s \) found that the next point is an \( r \)-point, then the source \( s \) decides that the next point on the path is on the other side from the destination point \( d \) and the algorithm continues. The above operation could also be applied for the destination \( d \) and it requires a constant number of steps and can be done in \( O(1) \) time. Figure 9 shows the generated path in bold in the case where \( s \) is within the convex hull of obstacle \( O_i \).

7 Concluding Remarks

In this paper, we proposed an algorithm to compute a collision-free path between a source and a destination in an environment with the existence of obstacles. We used the EREW R-Mesh as the underlying architecture. The algorithm was shown to run in constant time and requires \( O(\log N) \) preprocessing time. This outperforms the previous method that required a linear time in the number of obstacles with the same pre-processing time. We also proposed a method that could shorten the length of the path generated by the proposed algorithm.

One possible extension to this work is to design algorithms that get the shortest path between the source position and the destination position in the existence of obstacles. Other directions include enhancing the preprocessing time and using other parallel models to solve the path planning problem.

References