Enveloping Multi-Pocket Obstacles with Hexagonal Metamorphic Robots

Jennifer E. Walter  
Vassar College  
walter@cs.vassar.edu

Mary E. Brooks  
University of Oklahoma  
maryb@ou.edu

David F. Little  
Vassar College  
dalittle@vassar.edu

Nancy M. Amato  
Texas A&M University  
amato@cs.tamu.edu

Abstract—The problem addressed is reconfiguration planning for a metamorphic robotic system composed of any number of hexagonal robots when a single obstacle with multiple indentations or "pockets" is embedded in the goal environment. We extend our earlier work on filling a single pocket in an obstacle to the case where the obstacle surface may contain multiple pockets. The planning phase of our algorithm first determines whether the obstacle pockets provide sufficient clearance for module movement, i.e., whether the obstacle is "admissible". In this paper, we present algorithms that sequentially order individual pockets and order module placement inside each pocket. These algorithms ensure that every cell in each pocket is filled and that module deadlock and collision do not occur during reconfiguration. This paper also provides a complete overview of the planning stage that is executed prior to reconfiguration and presents a distributed reconfiguration schema for filling more than one obstacle pocket concurrently, followed by the envelopment of the entire obstacle. Lastly, we present examples of obstacles with multiple pockets that were successfully filled using our distributed reconfiguration simulator.

Index Terms—Metamorphic robots, hexagonal robots, obstacle, pocket, distributed reconfiguration

I. INTRODUCTION

A self-reconfigurable [5] robotic system is a collection of independently controlled, mobile robots, each of which has the ability to connect, disconnect, and move around adjacent robots. Metamorphic robotic systems [3], a subset of self-reconfigurable systems, have the following additional requirements:

- Each module (robot) is required to be identical in structure, motion constraints, and computing capabilities.
- Modules have a regular symmetry so that they can densely pack the plane to form two and three dimensional solid lattices.
- Modules are not independently mobile. They require a substrate lattice in order to achieve locomotion.

Shape changing in these composite systems is envisioned as a means to accomplish various tasks, such as structural support or tumor excision [7], as well as being useful in environments not amenable to direct human observation and control (e.g., interplanetary space, deep oceanic environments). The complete interchangeability of the robots provides a high degree of system fault tolerance, allowing the potential for system "self-repair" [6].

We focus on a system composed of planar, hexagonal robotic modules as described by Chirikjian [3]. Our strategy for motion planning begins with a centralized planning phase. The actual reconfiguration is distributed and relies on the assumption of global knowledge of the coordinates of cells in the goal configuration, G, local contact information at the modules, and that the initial configuration, I, is a straight chain configuration that intersects G. We have previously applied this approach to the problem of reconfiguring a straight chain to an intersecting straight chain [12] and to a goal configuration that satisfies a general "admissibility" condition [11]. In [10], we presented algorithms and heuristics to choose admissible substrate paths for module traversal that result in fast reconfiguration. Informally, an admissible substrate path is a sequence of contiguous cells that span the goal configuration in an east to west direction. This path forms a barrier between goal cells to the north and south, allowing efficient module traversal without collision or deadlock. In [9] we presented a definition of directionally-oriented admissible traversal surfaces, along with algorithms to plan and perform reconfiguration when a simple obstacle (i.e., one with no indentations or "pockets" in its surface) is embedded in G. We presented and analyzed algorithms to fill a single obstacle pocket in [8].

In this paper, we use the technique presented in [8] to determine whether an obstacle with multiple pockets embedded in G fulfills a simple admissibility requirement. Then we present algorithms to fill each pocket, both sequentially and concurrently.

A. Related work

The problem of enveloping obstacles in the goal environment with metamorphic robots has been addressed by some researchers. Chirikjian [3], for example, proposed a heuristic to attract deformable hexagonal modules to an obstacle in the goal configuration, causing the modules to converge around the obstacle as part of a centralized reconfiguration strategy. Bojinov et al. [1] provide a distributed strategy for grasping objects in the environment using rigid rhombic dodecahedral modules by probabilistically "growing" extensions to envelop the obstacle.

Other work addresses the locomotion of metamorphic robots over irregular traversal surfaces with "hills" or stair-like structures. Butler et al. [2] present a rule set for distributed locomotion of layers of deformable cubic modules....
over obstacles on the traversal surface. Hosokawa et al. [4] consider obstacles that form a surface such as a stairway to be traversed by rigid square modules.

B. Our approach and problem definition

Our overall objective is to design a distributed algorithm that will cause the modules to move from an initial straight chain configuration, \( I \), in the plane to a known goal configuration, \( G \). This algorithm should ensure that modules do not collide with each other, and the reconfiguration should be accomplished in a minimal number of rounds.

In this paper, we consider how to perform reconfiguration when an obstacle enveloped by the goal environment has multiple indentations, or pockets, in its surface. We first briefly review the techniques and algorithms presented in [8] to determine the admissibility of an obstacle, i.e., whether it is possible to fill pockets in the surface of the obstacle given the motion constraints on the robots, and to determine the order in which the modules will fill the goal cells in an admissible obstacle pocket. Then we describe algorithms to perform reconfiguration planning when an obstacle with multiple pockets is present in \( G \). This is the first paper, to our knowledge, to consider reconfiguration when the goal environment contains an obstacle with multiple pockets that are to be filled with modules.

In Section II we describe the system assumptions. Sections III and IV give an overview of our earlier work and describe how that work is extended to more complex obstacle shapes. Section V gives an overview of our distributed reconfiguration schema. We present examples of obstacles with multiple pockets that our algorithm successfully filled in simulation experiments in Section VI. Finally, Section VII provides a discussion of our results and future work.

II. SYSTEM MODEL

The plane is partitioned into equal-sized hexagonal cells and labeled using the same coordinate system as described by Chirikjian [3].

A. Assumptions about the modules

- Each module is identical in computing capability and runs the same program.
- Each module is a hexagon of the same size as the cells of the plane and always occupies exactly one of the cells.
- Each module knows at all times:
  - its location (the coordinates of the cell that it currently occupies),
  - its orientation (which edge is facing in which direction), and
  - which of its neighboring cells is occupied by another module.

Modules move according to the following rules.

1) Modules move in lockstep rounds.
2) In a round, a module \( M \) is capable of moving to an adjacent cell, \( C_1 \), iff (cf. Fig. 1)

(a) cells \( C_1 \) and \( C_2 \) are currently empty and
(b) module \( M \) has a neighbor \( S \) that does not move in the round (called the \textit{substrate}) and \( S \) is also adjacent to cell \( C_1 \).

3) Only one module tries to move into a particular cell in each round.
4) Modules cannot carry, push, or pull other modules, i.e., a module is only allowed to move itself.
5) Modules are deformable and move by a combination of rotation and changing joint angles.

Fig. 1. Before and after module movement. \( S \) is substrate module.

III. RECONFIGURATION PLANNING OVERVIEW

Motion constraints on the robots dictate the necessity of having at least two contiguous free sides facing the direction the robot will rotate. Another constraint on movement is imposed by our distributed algorithm, which forbids modules from moving if they determine locally that a move may partition the configuration. Fig. 2 shows the contact patterns of modules that are \textit{free} to move, those that are \textit{blocked} from moving by motion constraints, and those that may cause \textit{partitioning} of the system.

Fig. 2. Contact patterns for modules and pocket cells.

Our results in [12] and [9] show that maximum concurrency without deadlock can be ensured if moving modules are separated by two empty cells while moving over a surface that satisfies the properties of an admissible traversal surface. Informally, a north-facing (south-facing) traversal surface is \textit{east-monotone admissible} if
- modules will move exclusively from west to east over the surface.
- every clockwise (CW) or counter-clockwise (CCW) rotation is in a non-westerly direction, and
- modules separated by two empty cells can move over the surface without creating a deadlocked configuration.

Deadlock avoidance in our algorithm requires that vertical columns in the traversal surface that are traversed on the east side are separated from vertical columns that are traversed on the west side by at least three empty cells. We showed in [11] that our algorithm for reconfiguration will successfully fill any goal configuration that contains an east-monotone admissible traversal surface, also known as an admissible substrate path.

A. Reconfiguration planning with one obstacle in G

An obstacle is a sequence of one or more “forbidden cells” that modules cannot enter. We consider obstacles that are composed of hexagons of the same size as the cells of the plane and assume each hexagon in the obstacle occupies exactly one of the cells in the plane. Any indentation or “pocket” in the obstacle surface is assumed to contain goal cells. Informally, any goal cell found on a straight-line path between the center points of non-adjacent obstacle cells \( o_i \) and \( o_j \), such that the straight-line path is normal to one side of both \( o_i \) and \( o_j \) and passes through only non-obstacle cells, besides \( o_i \) and \( o_j \), is a pocket cell (cf. Fig. 3). Pocket cells that are adjacent to non-pocket goal cells are said to be on the pocket opening.

![Obstacle cell](image1)

Fig. 3. Pocket cells within obstacle. Arrows point to pocket opening cells.

If the obstacle pocket contains only cells with free contact patterns (cf. Fig. 2), we call the obstacle admissible. The free contact patterns in Fig. 2 represent non-occupied pocket cells with a sufficient number of contiguous non-occupied sides to allow module entry without partitioning the pocket. Inadmissible obstacles have pockets that contain cells with blocked or partitioning contact patterns. It is noteworthy that we use the same contact patterns during reconfiguration to locally distinguish a movable module as are used in the planning stages to determine whether an obstacle pocket can be filled.

Our method of enveloping an obstacle containing a single pocket in \( G \) requires the following planning steps:

(a) Choose an admissible substrate path that links the westmost column of \( G \) to the westmost obstacle column.

(b) Based on the choice of substrate path from part (a), choose a cell in the westmost column of \( G \) as an intersection point for the straight chain of modules in \( I \) and position \( I \) so that it is collinear with the substrate path.

(c) Determine the order in which modules will fill the obstacle pocket.

(d) If the obstacle/pocket mass has any east-facing vertical sections after the pocket coordinates have been ordered, determine the coordinates of goal cells that will be filled by modules to extend the obstacle so that it tapers gradually to the east.

(e) Choose an admissible substrate path that links the eastmost column of the extended obstacle to the eastmost column of \( G \) if necessary.

(f) Determine the order to fill cells to the north and south of the substrate path/obstacle mass in vertical columns from east to west.

Planning steps (a), (b), (c), and (f) were the subject of [10], [11] and [9]. We now briefly review steps (e) and (d), the planning algorithms to fill a single obstacle pocket and to taper the obstacle to the east.

1) Filling a single obstacle pocket: Admissible obstacle pockets may not be east-monotone admissible traversal surfaces, meaning that modules traversing a pocket in an obstacle surface may enter deadlocked configurations inside the pocket. However, we require only that modules crawl into, not back out of pockets. We presented an algorithm in [8], called FILLPOCKET (cf. Fig. 4), to determine the order in which modules should line the inside of a pocket to ensure that no unoccupied portions of the pocket are blocked. This algorithm returns a list of pocket cells containing the coordinate sequence for filling cells in the pocket.

```
Algorithm FILLPOCKET(pocket[i], entryCell)
1. Create perimeter array, starting at entryCell
2. While ([perimeter] > 0)
3.   \( z \) = cell with lowest number of free sides in
4.   deepest NCPP range, starting search
5.   at perimeter[1]
6.   \( z \) = add \( z \) to tail of returnList
7.   Remove \( z \) from perimeter after removing \( z \)
8.   \( z \) = add appropriate neighbor of \( z \) to perimeter
9. Adjust number of free sides on \( z \)’s neighbors
10. End While
11. return returnList
```

Fig. 4. Pseudocode for Algorithm FILLPOCKET.

Rotation for modules directing the filling pocket is determined prior to the execution of FILLPOCKET by calculating whether the closest pocket opening cell is in the CW or CCW direction. Here, proximity is measured in terms of traversal distance over the surface of the obstacle, from the cell where the substrate path meets the obstacle. All modules filling a single pocket must be rotating the same direction.

Any pocket cell that is initially adjacent to an obstacle cell is designated a perimeter cell by the FILLPOCKET algorithm. Fig. 5 shows an obstacle pocket in which the perimeter cells have been numbered consecutively in the order modules would traverse the interior of the pocket when rotating CCW, starting at the cell numbered 1. The algorithm uses perimeter cell contact information and the discovery of non-consecutive perimeter pairs (NCPs), i.e., adjacent perimeter cells that are not numbered consecutively, to determine the order to fill
the cells in an obstacle pocket. An NCPP indicates a portion of the pocket in which there are only 2 empty cells separating occupied cells.

In step 3 of Fig. 4, the FillPocket algorithm starts at cell 1 on the pocket perimeter and proceeds sequentially through the perimeter cells, from lowest to highest, looking for an NCPP. When an NCPP is found, the algorithm continues searching only within the range of the cells delimiting the most recent NCPP found, for a “deeper” NCPP. After a cell is placed in the ordering, or “filled”, it is treated like an obstacle cell; the perimeter is renumbered around the filled cell and the algorithm begins looking for the next cell to fill, starting again at cell 1 on the pocket perimeter. If no NCPPs are found within the range of the cells delimiting the most recent NCPP found, the algorithm chooses a perimeter cell with the lowest number of free sides within that range to be filled next. If the pocket contains no NCPPs, the perimeter cell with the lowest number of free sides is chosen to be filled. This process is repeated until all pocket cells are included in the ordering.

We proved in [8] that algorithm FillPocket will produce an ordering that includes every pocket cell in a single-pocket admissible obstacle and that all pocket cells will eventually be filled by modules.

2) Extending or “repairing” an obstacle: In the last phase of reconfiguration, modules fill the portions of the goal to the north and south of the substrate path/obstacle mass in vertical columns from east to west. During this process, narrow gaps may form between a vertical column of obstacle cells and the vertical columns of newly occupied goal cells. Our “repair” algorithm, presented in [9], prevents such deadlock situations by ensuring that the obstacle tapers gradually to the east.

IV. RECONFIGURATION PLANNING FOR MULTI-POCKET OBSTACLES

Just as in the single pocket case, each pocket cell in an admissible obstacle with multiple pockets must have a free contact pattern (cf. Fig. 2) or some goal cells will not be filled. This section presents algorithms to count the number of pockets, determine the order in which pocket cells will be filled, and calculate module rotation direction for the modules designated to fill each pocket.

A. Counting pockets

If the obstacle is admissible, we map a graph \( P \) onto the pocket cells, such that there is a vertex centered in each pocket cell, with edges between vertices of neighboring pocket cells. Each node in \( P \) is a pocketCell object containing the coordinates of the pocket cell it represents, current contact patterns of that pocket cell, and fields for the rotation direction, coordinates of the entry cell for modules entering the pocket, and current perimeter number of the cell. The perimeter number and the current contact patterns are used in algorithm FillPocket, from Fig. 4, and will not be discussed further in this paper. The other variables are named and discussed as they are used in the algorithms presented below.

In algorithm CountPockets, shown in Fig. 6, we use breadth-first search on the set of pocket cell objects in the graph \( P \) to group the objects by individual pockets and also to count the number of pockets. A vector called pocketCells initially contains every pocketCell object in the obstacle. The first pocketCell object \( x \) in pocketCells is chosen as the starting point for the search. Objects are removed from the pocketCells vector as they are discovered during the breadth-first search and are added to a vector of vectors called pocket, where each vector pocket(i) contains the objects representing all cells from a single pocket. A FIFO (first in, first out) queue (called \( Q \) in Fig. 6) holds the objects during the breadth-first search. After exhausting breadth-first search from pocketCell \( x \), if the pocketCells vector is not empty, breadth-first search is run again from the first pocketCell of those that remain in the pocketCells vector. This process is repeated until vector pocketCells is empty. Since \( i \) is incremented each time another breadth-first search is started, when pocketCells is empty, each vector pocket(i) will contain a set of objects corresponding to all the pocket cells in a single pocket.

Algorithm CountPockets

```plaintext
1. i := 0
2. While (pocketCells is not empty)
3.   x := pocketCells(i)
4.   i := i + 1
5.   remove x from pocketCells
6.   Q := queue
7.   While (Q is not empty)
8.     u := Q.dequeue
9.     For each neighbor v of u: v \in pocketCells
10.    Q := queue
11.   remove u from pocketCells
12.   end for
13.   end while
14.   end while
```

Fig. 6. Pseudocode for Algorithm CountPockets (initially, all pocket cells are in vector pocketCells).
### B. Directing and ordering cells in each pocket

Let startCell be the goal cell on the western substrate path that is adjacent to an obstacle cell. Once the number of pockets in the obstacle is known and the coordinates of the cells in each pocket have been distinguished, we sort the pocket vector according to the CW traversal distance over the obstacle surface from startCell to the closest pocket entry cell in each pocket vector. Fig. 7 shows how obstacle pockets would be ordered in a sample configuration. The coordinates of the closest pocket entry cell in the CW direction for pocket(i) are saved in each pocketCell object in pocket(i) as cwEntryCell and the coordinates of the closest pocket entry cell in the CCW direction are saved as ccwEntryCell.

Let k be the number of pockets found by COUNTPOCKETS in the obstacle. Let $d_{cw}^{1}, d_{cw}^{2}, \ldots, d_{cw}^{k}$ be the respective traversal distances across the surface of the obstacle in the CW direction from startCell to pockets pocket(1), pocket(2), ... pocket(k). Let $d_{ccw}^{1}, d_{ccw}^{2}, \ldots, d_{ccw}^{k}$ be the respective traversal distances across the surface of the obstacle in the CCW direction from startCell to pockets pocket(1), pocket(2), ... pocket(k). Let $num_{cw}^{1}, num_{cw}^{2}, \ldots, num_{cw}^{k}$ be the number of pocket cells that must be filled in the CW direction before modules rotating CW can reach pocket(i) from startCell and let $num_{ccw}^{1}, num_{ccw}^{2}, \ldots, num_{ccw}^{k}$ be the number of pocket cells that must be filled in the CCW direction before modules rotating CCW can reach pocket(i) from startCell. Let cwd and ccwd be arrays of length k such that $cwd[i] = d_{cw}^{i} + num_{cw}^{i}$ and $ccwd[i] = d_{ccw}^{i} + num_{ccw}^{i}$, for all $1 \leq i \leq k$.

![Fig. 7. Obstacle with three pockets, numbered in order of CW distance from startCell. Arrows represent total cells counted when measuring distance in CW and CCW directions from startCell to closest cell on each pocket opening.](image)

The direction modules will rotate to fill pocket(i) and the order in which the set of pockets will be filled is calculated by algorithm DIRECT&ORDER, shown in Fig. 8. As the cells are ordered, the rotation direction variable, rdir, in each object is set to either CW or CCW. In lines 4b and 5b, cell objects are added to the tail of a pair of singly-linked lists CWList and CCWList to order the modules that will fill pocket cells in the CW and CCW directions, respectively. This algorithm orders modules so that they will always rotate toward the closest pocket opening.

For the configuration shown in Fig. 7, cwd[1] = 5 and ccwd[1] = 28, so cells of pocket(1) would be added to CWList. For pocket(2), cwd[2] = 20 and ccwd[2] = 17, so pocket(2) cells would be added, in the order determined by algorithm FILLPOCKET, to CCWList. For pocket(3), cwd[3] = 31 and ccwd[3] = 6, so pocket(3) cells would be added, in the order determined by algorithm FILLPOCKET, to CCWList.

### C. Pocket filling strategies

Our first approach to filling each pocket is to completely fill all pockets in the CW direction, followed by all cells in the CCW direction. We use an initially empty, singly-linked list, called PocketList, to hold the coordinates of cells in the order they will be filled by modules. In this approach, PocketList contains all the cells in CWList, followed by all the cells in CCWList. Fig. 9 shows the order the cells in the triple-pocket obstacle of Fig. 7 are filled using this strategy.

![Fig. 9. Obstacle from Fig. 7, with pocket cells numbered in the order they would be filled by modules sequentially filling pockets in order found by algorithm DIRECT&ORDER.](image)

If there are pockets to be filled in both the CW and CCW directions, modules starting from positions in $i$ can alternate rotation direction during reconfiguration until all pocket cells in either the CW or CCW direction are filled, as described algorithmically in Fig. 10. This strategy maximizes parallelism during reconfiguration because alternating module rotation direction involves less delay to maintain the necessary spacing between moving modules than if neighboring modules in $i$ initially rotate in the same direction.

For example, using algorithm CONCURRENTFILL on the obstacle in Fig. 7, modules rotating CW could fill pocket $p_1$ while modules rotating CCW filled $p_2$ and $p_2$. Fig. 11 shows the order the cells in the triple-pocket obstacle of Fig. 7...
Algorithm CONCURRENTFILL
1. While (CWLList.IsEmpty() & & CWLList.IsEmpty())
   Remove head of CWList and add to tail of PocketList
   Remove head of CCWList and add to tail of PocketList
2. Add rest of cells in CWList or CCWList to tail of PocketList
Fig. 10. Algorithm to order pockets so that they are filled concurrently.

would be filled when pockets are filled concurrently using algorithm FILLPOCKET to order modules in each pocket.

Fig. 11. Obstacle from Fig. 7 with pocket cells numbered in the order they would be filled by modules alternating CW and CCW directions.

Both of the approaches above ensure that modules will not attempt to traverse an unfilled pocket and that modules rotating CW and CCW will not collide, a statement we prove below. The proof of Claim 1 shows that, due to the way the distance to each pocket is calculated, modules filling separate obstacle pockets run no risk of collision and modules will not attempt to traverse an unfilled pocket.

Claim 1: If pocket(i) is filled from the CW (CCW) direction, then so is pocket(i - 1) (pocket(i + 1)).

Proof: Since d_{cw} and num_{cw} are strictly decreasing and d_{cw} and num_{cw} are strictly increasing as i increases, if d_{cw} + num_{cw} ≤ d_{cw} + num_{cw}, then d_{cw} + num_{cw} ≤ d_{cw} + num_{cw}. Likewise, since d_{cw} and num_{cw} are strictly increasing and d_{cw} and num_{cw} are strictly decreasing as i increases, if d_{cw} + num_{cw} > d_{cw} + num_{cw}, then d_{cw} + num_{cw} > d_{cw} + num_{cw}.

Therefore, the modules will fill the obstacle pockets in order of increasing distance from startCell. Additionally, the surface of the obstacle perimeter traversed by modules rotating CW (CCW) will not intersect the surface of the obstacle perimeter traversed by modules rotating CCW (CW). Thus, modules filling separate pockets in the order calculated by algorithm DIRECT & ORDER will not cross paths as the pockets are being filled concurrently, nor will modules attempt to traverse unfilled pockets.

V. DISTRIBUTED RECONFIGURATION SCHEMA
The overall reconfiguration proceeds as outlined below. If G and the obstacle contained in G are admissible, fill in cells of G in the following order:

a) western substrate path, from west to east,
b) pocket cells in obstacle, in order listed in PocketList,
c) repaired cells north and south of obstacle, from west to east,
d) repaired cells east of obstacle, from west to east,
e) eastern substrate path, from west to east, and
f) north and south of substrate path and obstacle, from east to west.

Else report inadmissible configuration.

VI. SIMULATION RESULTS
We developed an object-oriented discrete event simulator to test the performance of our multi-pocket filling algorithm on a number of different obstacles embedded in G, including those shown in Fig. 12 and the obstacle shown in Fig. 7. In each of these cases and for every admissible obstacle tested, all pockets were filled successfully.

Fig. 12. Examples of obstacle pockets that were successfully filled by our algorithm. Shaded cells represent obstacle cells embedded in G and other goal cells are not shown.

VII. CONCLUSIONS AND FUTURE WORK
We have presented algorithms to plan reconfiguration of a system of hexagonal metamorphic robots when an obstacle embedded in the goal configuration has multiple pockets in its surface. The algorithms in this paper build on and extend our earlier work by increasing the set of obstacles that can be completely enveloped during reconfiguration.

We are currently working on relaxing the restriction that obstacle pockets be admissible, i.e., that all pocket cells have a free contact pattern (cf. Fig. 2).

REFERENCES