Incremental Multi-Scale Search Algorithm for Dynamic Path Planning With Low Worst-Case Complexity

Yibiao Lu, Xiaoming Huo, *Senior Member, IEEE*, Oktay Arslan, and Panagiotis Tsiotras, *Senior Member, IEEE*

***Abstract*—Path-planning (equivalently, path-finding) problems are fundamental in many applications, such as transportation, VLSI design, robot navigation, and many more. In this paper, we consider dynamic shortest path-planning problems on a graph with a single endpoint pair and with potentially changing edge weights over time. Several algorithms exist in the literature that solve this problem, notably among them the Lifelong Planning A* (LPA*) algorithm. The LPA* algorithm is an incremental search algorithm that replans the path when there are changes in the environment. In numerical experiments, however, it was observed that the performance of LPA* is sensitive in the number of vertex expansions required to update the graph when an edge weight value changes or when a vertex is added or deleted. Although, in most cases, the classical LPA* requires a relatively small number of updates, in some other cases the amount of work required by the LPA* to find the optimal path can be overwhelming. To address this issue, in this paper, we propose an extension of the baseline LPA* algorithm, by making efficient use of a multiscale representation of the environment. This multiscale representation allows one to quickly localize the changed edges, and subsequently update the priority queue efficiently. This incremental multiscale LPA* (m-LPA* for short) algorithm leads to an improvement both in terms of robustness and computational complexity–in the worst case–when compared to the classical LPA*. Numerical experiments validate the aforementioned claims.**

***Index Terms*—A* algorithm, beamlet-like structure, dynamic programming, LPA* algorithm, path-planning, quadtrees.**

I. Introduction

DYNAMIC path-planning deals with the solution of shortest-path problems on a graph, when the edge weights in the graph change over time. The Lifelong Planning A* algorithm (or LPA* for short) [1] is well-known and widely used algorithm to solve dynamic path-planning problems, especially in mobile robotic applications. In numerical experiments it was observed that LPA* can have unfavorable worst-case complexity. In particular, if a vertex located close to the original optimal path changes, a large number of vertex expansions may be required to replan the optimal path to the destination. In other words, LPA* is sensitive (i.e., not "robust"), in the sense that the number of required vertex updates can vary widely, depending on the location of the updated vertex in the graph. To demonstrate this point, consider the dynamic shortest-path problem illustrated in Fig. 5. The objective in this figure is to find the shortest path while some of the vertices may become blocked over time. In Fig. 6 (in blue line), we observe that the number of vertex expansions in the LPA* varies significantly for this problem. Although most of the time, LPA* requires only a few expansions, in several noticeable occasions, the number of expansions is huge. Such a variation in the number of vertex expansions is not desirable, as it reveals unfavorable worst-case performance. Ideally, one would like the number of expansions to be relatively immune to the location of the updated vertices. Our objective is to introduce a modification of the LPA* that keeps the number of expanded vertices approximately constant (compared to the classical LPA* implementation), regardless of the location of the updated vertex.

The main idea of the proposed *multiscale* LPA^* $(m\text{-}LPA^*)$ algorithm is to utilize pre-computed multiscale information of the environment to formulate an associated search graph of smaller size, therefore reducing the computational complexity. The intuition behind the proposed algorithm is summarized as follows. Consider a uniform n-by-n grid representing the world (or an image) assuming, without loss of generality, four-nearest-neighbor connectivity. Given a source and a destination, the search graph is abstracted from the environment before applying any path-planning search algorithm. When a change in the environment leads to the update of a certain vertex in the graph, the amount of computations required by LPA* during replanning varies dramatically, depending on the location of the updated vertex. Specifically, when the update happens to induce a "local dead-end" or if it is near the source, the number of vertex expansions can be extremely high. Fig. 6 demonstrates this point. The proposed m-LPA* algorithm, on the other hand, takes advantage of multiscale information extracted from the environment and therefore reduces the computational complexity in both the initial planning and replanning steps simultaneously. This is achieved by making extensive use of a beamlet-like graph structure, which is based on a suitably pruned quadtree representation of the environment (called in the sequel as path finding reduced recursive dyadic

Manuscript received June 8, 2010; revised November 17, 2010, February 16, 2011, and May 11, 2011; accepted May 12, 2011. Date of publication June 16, 2011; date of current version November 18, 2011. This paper was recommended by Associate Editor X. Wang.

Y. Lu and X. Huo are with the H. Milton Stewart School of Industrial and System Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: ylv3@gatech.edu; huo@gatech.edu).

O. Arslan and P. Tsiotras is with the D. Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: oktay@gatech.edu; tsiotras@gatech.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCB.2011.2157493

partitioning or PFR-RDP). The PFR-RDP encodes the information of the environment in a hierarchical, multiscale fashion, keeping track of "long-range" interactions between the vertices in the underlying beamlet graph. Therefore, when a vertex update that would induce a "local-dead-end" in the nearest neighbor graph takes place, this information is "transmitted" in the beamlet graph by modifying the hierarchical structure of the PFR-RDP to include new vertices and edges, so that no more dead-end exists in the new (beamlet) graph.

In the initial planning step, m-LPA* works exactly the same way as the previously proposed m-A* algorithm [2], by formulating a smaller-size search graph from the multiscale information obtained from a specifically designed bottom-up fusion algorithm (see Algorithm 2 in Section IV-B). This leads to a significant reduction in computational complexity (see Section III for a brief review of the A* and m-A* algorithms). When a vertex is updated, m-LPA* will further decompose the area where the updated vertex is located and will update the graph before replanning. The number of vertex expansions during the replanning step is therefore reduced, owing to the smaller-size search graph obtained in the initial planning.

The theoretical analysis of the associated computational complexity reveals that, in the worst case, the proposed algorithm has a lower order of complexity than the LPA* algorithm. In our numerical experiments, it was found that the m-LPA* algorithm can dramatically reduce the number of vertex expansions, in the worst case. To ensure a fair comparison, the implementation of both LPA* and m-LPA* algorithms was based on Fibonacci heaps, which is known to be one of the most efficient data structures for sorting problems [3]. We believe that a comparison using Fibonacci heaps is more accurate than using a binomial heaps [1]. In addition, the use of Fibonacci heaps allows one to find the minimum vertex in the priority queue faster. Based on these numerical studies, it is shown that the proposed method is very stable when tested under several different scenarios.

The proposed algorithm belongs to the general class of multiscale/multiresolution, dynamic path-planning algorithms. Multiresolution decomposition techniques for path-planning have been used extensively in the literature. See, for example, [4]–[8] and references [9]–[11] where wavelets that are used to create several levels of abstraction for the environment. The approach in [10], [11] uses a higher resolution close to the agent where is needed most, and a coarser resolution at large distances from the current location of the agent. The motivation for this approach stems from the fact that the agent's immediate reaction to an obstacle or a threat is needed only in the vicinity of its current position. Faraway obstacles or threats do not have a great impact on the vehicle's *immediate* motion. Therefore, it makes sense from a computational point of view to generate a solution with greater accuracy only locally, around the current location of the agent, with decreasing resolution further away. Multiresolution strategies are indispensable for on-line implementation when the agent has limited computational resources. In this paper, we go beyond the use of wavelets to encode the environment information across different scales. Although wavelets are very efficient in that respect, they also have some drawbacks. Most importantly, they lack orientation information and can be roughly characterized as isotropic. Beamlets provide a framework for multiscale analysis, in which line segments play a role analogous to the role played by points in wavelet analysis. They add two crucial elements missing from wavelet processing: 1) orientation; and 2) elongation information. They provide an optimal way to approximate curvelinear features in 2-D. Beamlets connect points that may be far apart, thus encoding "far away" interactions in the environment. In this paper, we show that multiresolution/multiscale strategies based on beamlet-like ideas, can also lead to increased computational robustness at the execution level.

The rest of the paper is organized as follows. 1) Section II formulates the path-planning problem in a dynamically changing environment; 2) Section III reviews the multiscale A* algorithm and the LPA* algorithm. These two algorithms are the foundation of the m-LPA* algorithm proposed in this paper; 3) Section IV offers the details of the proposed m-LPA* algorithm. Its complexity analysis is conducted in Section V; 4) Section VI provides numerical examples to compare m-LPA* with LPA*; 5) while Section VII compares m-LPA* with other closely related search algorithms, pinpointing their apparent similarities and their main differences; and 6) Section VIII summarizes the findings of the paper and suggests some possible future extensions.

II. Problem Formulation

In a path-planning problem we are given a graph $G = (V, E)$, with vertex set V and edge set $E \subseteq V \times V$. For instance, V is typically the set of possible vehicle locations and E represents transitions between these locations. The weight of each edge denotes the cost of transitioning between the two locations represented by the vertices connected by the corresponding edge. Planning a path in G can thus be cast as a single-pair shortest path problem on this graph.

In a deterministic environment, a graph has constant edge weights over time. Dijkstra's algorithm and the A* algorithm and their numerous variants are classical methods to compute optimal paths to the destination from every location or from a single location in the graph, respectively. In many practical applications (especially in the area of robotic mobile vehicles) it is common to obtain updated information about the environment over time. This leads to a replanning problem on a graph with changing edge weights. If the replanning is done from scratch (that is, without using any prior information of the graph structure or the edge weights), this may result in wasted time and/or computational resources. Therefore, several efficient algorithms have been developed in the literature to adjust the current shortest path when a change takes place, without performing redundant calculations. References [1] and [12], in particular, proposed the Lifelong Planning A* (LPA*) algorithm which considers the dynamic path-planning problem with fixed source and destination. Another popular replanning algorithm for a vehicle navigating in a dynamically changing environment that combines heuristic and incremental searches is D* (or dynamic A*) [13]. In [14] the authors extended the LPA* algorithm and proposed the D*-Lite algorithm, which is similar in spirit, but much simpler than D*. In many applications,

the curvature of the resulting path is also of interest in order to guarantee kinematically feasible vehicle trajectories. In this context, references [15]–[18] incorporate curvature information to make the planned route as realistic as possible.

In this paper, we consider a path-planning problem in a dynamically changing 2-D environment. Without loss of generality, it is assumed that the environment can be represented by an n-by-n square image, where n is dyadic, i.e. $n = 2^J$ and J is a positive integer. Note that such an image-based formulation is quite common in the path-planning literature [15]. It is assumed that the image contains two types of pixels: 1) gray pixels (representing non-traversable obstacles); and 2) white pixels (representing traversable free cells). The path-planning problem is to find the shortest path between a given pair of *source* and *destination* pixels. Under this binary image assumption, a change in the environment is formulated as a change in the *traversability properties* between certain cells.[1] The task of replanning then consists of finding another shortest path inside the new environment, while avoiding a large number of further vertex expansions.

Among the several existing replanning algorithms mentioned previously, in this paper we focus exclusively on LPA* since it represents a widely used, state-of-the-art algorithm in the area of incremental replanning. It is reminded that LPA* operates essentially the same as the well-known A* algorithm in the initial planning step but, in addition, it utilizes the concept of "local inconsistency" of vertices to control the size of the priority queue and hence the number of vertex expansions.

As mentioned earlier, numerical examples indicate that the number of vertex expansions in the LPA* algorithm can vary significantly depending on the obstacle location. Specifically, in several cases when the blocking vertex happens to be somewhere along the initial shortest-path resulting in a "local dead end," or when there are too many obstacles in the environment, the number of vertex expansions in the LPA* replanning part can blow up. We want to avoid such undesirable behavior. Our objective is to devise an algorithm that retains the nice properties of LPA*, while at the same time it maintains an almost constant number of vertex expansions when the environment is changed. The proposed multiscale LPA* (m-LPA*) algorithm takes advantage of the sparse information induced by the quadtree decomposition in a hierarchy of dyadic squares. This is the same technique adopted in our previously proposed *multiscale* A^* (m-A^*) algorithm [2]. As shown in Section V, such a strategy can reduce the number of vertex expansions significantly, and therefore also reduce the overall worst-case replanning complexity.

III. Multiscale A* and Lifelong Planning A* Algorithms

In this section we briefly summarize the key points of the multiscale A* algorithm (m-A*), which lay the foundation for m-LPA*. The m-A* algorithm is an extension of the classical A* algorithm; it takes advantage of preprocessed multiscale information in order to reduce the overall computational complexity. Afterwards, we provide a brief overview of the LPA* algorithm. The LPA* is an incremental search algorithm that replans the initial path when a vertex update occurs owing to a change in the environment. These two algorithms (the m-A* and the LPA*) are the precursors of m-LPA*.

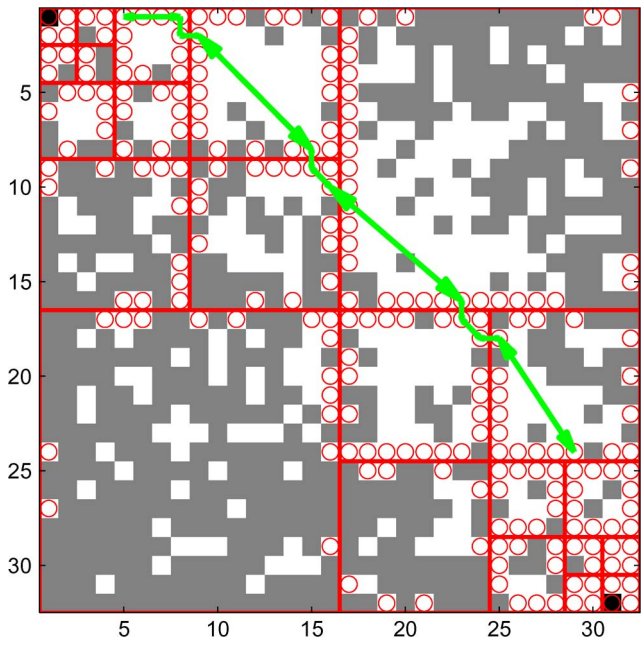


Fig. 1. Illustration of the path-finding reduced recursive dyadic partition (PFR-RDP) on a 32 × 32 image. The black cells are the source and destination. The red circles denote the free boundary cells (i.e., the vertices in the beamlet graph). The green arrows show the edge weights between two free cells constructed via the bottom-up fusion algorithm across several scales in the PFR-RDP.

A. Multiscale A^* (m-A^*) Algorithm

The classical A* algorithm searches through all free cells in the environment, which can be overwhelmingly redundant. The motivation for the multiscale-A* (m-A*) algorithm is to construct a smaller size search graph, on which the computational complexity of searching for the shortest path is significantly reduced. The m-A* algorithm is based on the following key elements:

(i) The *recursive dyadic partition* (RDP) and its extension, the *path-finding reduced RDP* (PFR-RDP). The elements of (PFR-)RDP are the dyadic *d-squares*, parameterized by *scale* and *location*. For the single-pair, shortest path-planning problem, the PFR-RDP is first constructed, and all free cells at the boundaries of each d-square in the PFR-RDP are selected as the vertices in the search graph. Fig. 1 shows the d-squares for the shown partition, along with the additional boundary cells used as vertices in the search graph.

(ii) The idea of *beamlet-like connectivity*. This provides an extension of connectivity between a pair of non-adjacent free cells. Within each d-square, besides the assumption of four-nearest-neighbor connectivity, we further consider any pair of free boundary cells to be connected if there exists an obstacle-free path between the two, which

[1] A cell is defined to be a collection of pixels in the environment. At the highest resolution, cell and pixels are equivalent.

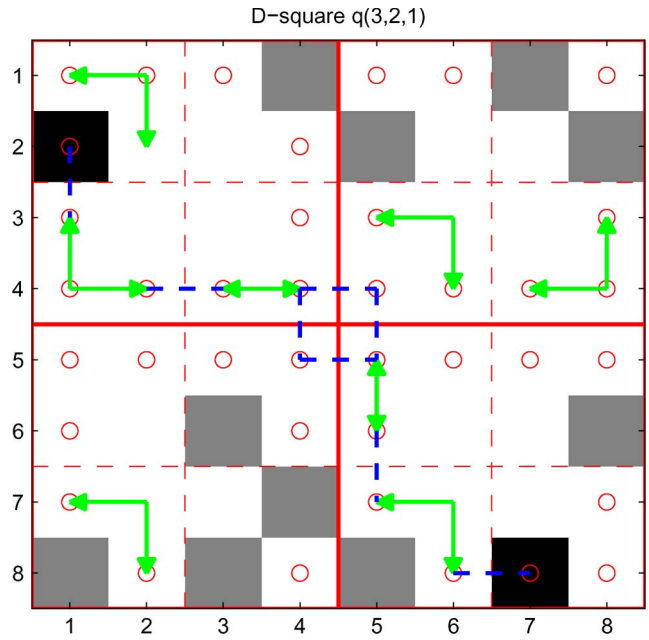


Fig. 2. Magnified view of the fusion process in the d-square q(3,2,1) of Fig. 1. The fusion is conducted in a complete dyadic partition. Notice that the solid red grid stands for the partition corresponding to the second layer of the associated quadtree and the dashed red grid corresponds to the partition with respect to the third layer. The green arrow lines indicate the inter-distance connectivity between the corresponding free boundary cells. The blue lines show the fusion process.

lies within that d-square. This shortest path is called a *beamlet*. Readers may notice that this is a generalization of the beamlet concept introduced in [19]. Therein, the beamlets are defined as straight line segments of variable length, scale and angle, connecting boundary cells of d-squares. They have been applied successfully to image processing applications (i.e., edge detection). Please see [20]–[22] for more details. The *beamlet graph* is defined to be the search graph with these two types of connectivity.

(iii) The *bottom-up fusion algorithm* designed to obtain the edge weights of the search graph from different scale dyadic squares. The algorithm is a recursive method that employs the RDP in each d-square from the PFR-RDP, in order to compute the inter-distances between the free boundary cells for all d-squares in the environment. In other words, the bottom-up fusion algorithm finds all shortest paths between the free boundary cells (i.e., the edge weights in the beamlet graph). The main idea of the algorithm is based on the observation that if we know the inter-distances between the free boundary cells within each of the smaller d-squares, and by considering the connectivity of the free boundary cells that belong to neighboring d-squares, we can treat all free boundary cells of the four d-squares at the next scale as vertices in a "fused" graph. The distance between free cells from neighboring d-squares can be defined by direct neighbors. Fig. 2 shows how this "fusion" of distance can be conducted recursively. The pseudo-code of the bottom-up fusion algorithm is given in Algorithm 2.

The combination of beamlet-like connectivity and multiscale decomposition in m-A* can reduce the depth of the search tree from $O(n)$ roughly to $O(\log n)$, without increasing the branching factor in each layer. The beamlet graph thus has $O(n)$ vertices and $O(n^2)$ edges. The worst-case complexity of running A* on the beamlet graph is therefore $O(n^2)$, compared to $O(n^2 \log n)$ when using the four-nearest-neighbor graph.

B. Incremental Search Algorithm: LPA^*

Heuristic search methods are likely to find the shortest path faster than uninformed search methods. Incremental search methods, on the other hand, promise to find shortest paths by solving a series of similar path-planning problems faster than it is possible by solving each path-planning problem from scratch. The Lifelong Planning A^* (LPA^*) in [1] can be viewed as an incremental version of the A* algorithm, the latter being a heuristic enhancement of the well-known Dijkstra algorithm. The LPA* repeatedly finds shortest paths from a given source to a given destination, while the edge weights of the graph change, or while vertices are added or deleted. The first search of LPA* is the same as that of the classical A* algorithm. Subsequently, the algorithm breaks ties in favor of vertices with a smaller g-value (i.e., the current estimated distance from the start vertex). As a result, many of the subsequent searches are potentially faster, because the algorithm reuses those parts of the previous search graph that are identical to the new one.

To facilitate the subsequent discussion on the proposed multiscale version of LPA*, we first need to introduce some notation as follows. Let $Succ(v) \subset V$ denote the set of successors of the vertex $v \in V$, let $Pred(v) \subset V$ denote the set of predecessors of vertex v, and let $c(v, v')$ denote the cost of moving from vertex v to vertex v'. Let also $h(v, v_{\text{goal}})$ denote the heuristic that guides the search direction to the goal destination. Finally, let $g(v)$ denote the start distance of vertex v, that is, the length of the shortest path from v_{start} to v.

The LPA* algorithm uses two estimates of the start distance, namely: 1) $g(v)$; and 2) $rhs(v)$. The rhs-values are the one-step lookahead values based on the g-values for each vertex, and thus they are potentially better informed than the g-value. Specifically, $rhs(v) = \min_{v' \in Pred(v)}(g(v') + c(v', v))$. A vertex is defined to be *locally consistent* if and only if $g(v) = rhs(v)$; otherwise it is said to be locally inconsistent. It should be clear from the above definitions that all vertices are locally consistent if and only if their g-values are equal to their respective start distances. LPA* also maintains a priority queue, which always contains exactly the locally inconsistent vertices. The priority of vertices in the queue is based on the key value, which is defined to be $k(v) = [k_1(v), k_2(v)]$, where $k_1(v) = \min(g(v), rhs(v)) + h(v, v_{\text{goal}})$, and $k_2(v) = \min(g(v), rhs(v))$. The keys of the vertices in the priority queue roughly correspond to the f-values used by A*. LPA* always recalculates the g-value of the vertex (i.e., expands the vertex) in the priority queue with the smallest key value. This is similar to A*, which always expands the vertex in the priority queue with the smallest f-value. LPA* keeps expanding the vertices until v_{goal} is locally consistent and the key of the vertex to be expanded next is no less than the key of v_{goal}. This is

similar to A*, which expands vertices until it expands v_{goal}, at which point the g-value of v_{goal} equals its start distance, and the f-value of the vertex to be expanded next is no less than the f-value of v_{goal}. If $g(v_{\text{goal}}) = \infty$ after the search, then there is no finite-cost path from v_{start} to v_{goal}.

Note that LPA* does not make every cell locally consistent. Instead, it uses an informed heuristic to focus the search and the subsequent updates only on the vertices whose g-values are relevant for finding the shortest path. This is the main principle behind LPA*, and this is what makes LPA* a very efficient replanning algorithm.

IV. Multiscale Strategy in Dynamic Path Planning: m-LPA*

The proposed multiscale Lifelong Planning A* (m-LPA*) algorithm is an extension of the previously proposed m-A* algorithm [2] for the case of a dynamically changing environment. Recall that we consider a discrete 2-D environment of dimension $n \times n$ containing only *obstacles* and *free cells*. Each free cell is a vertex in the underlying topological graph and is connected to the free cells among its four nearest-neighbors. We assume, without loss of generality, that each edge has unit weight. We describe our approach in three steps. In Section IV-A, we describe the *dynamic path-finding reduced recursive dyadic partition* (DPFR-RDP) step. This serves as the starting point of the proposed replanning scheme. We then describe how to update the beamlet graph for the purpose of replanning when a change that alters the traversability of the original path takes place. This is done in Section IV-B. The approach hinges upon the bottom-up fusion algorithm that collects the multiscale information of the graph. Section IV-C presents the proposed multiscale LPA* algorithm, which essentially runs LPA* iteratively on the aforementioned updated beamlet graph.

A. Dynamic Path-Finding Reduced Recursive Dyadic Partition

We define two types of recursive dyadic partitions (RDP), namely, the complete RDP and the path-finding reduced RDP (PFR-RDP). The PFR-RDP is a partial recursive partition in the sense that not all d-squares are partitioned to the finest level. Fig. 1 shows an example of a PFR-RDP on a 32-by-32 image.

In order to make efficient use of the multiscale information in a dynamically changing environment, we extend this recursive dyadic partition to obtain a dynamic version of PFR-RDP: the Dynamic PFR-RDP (DPFR-RDP). This is just one more step in the construction of the PFR-RDP, in the sense that when a certain cell in the gridworld suffers from a traversability change, we first identify the d-square in the PFR-RDP in which this candidate cell is located, and we then conduct a further partial dyadic partition (only) in this candidate d-square. Fig. 3 shows the DPFR-RDP for a 64-by-64 image. The pseudo code for the DPFR-RDP is given in Algorithm 1.

The DPFR-RDP enables us to obtain high resolution information around the cells in the environment that have changed. All free boundary cells for each d-square obtained from further dyadic partitioning are included as new vertices in the beamlet graph. This is the vertex update step of the beamlet graph.

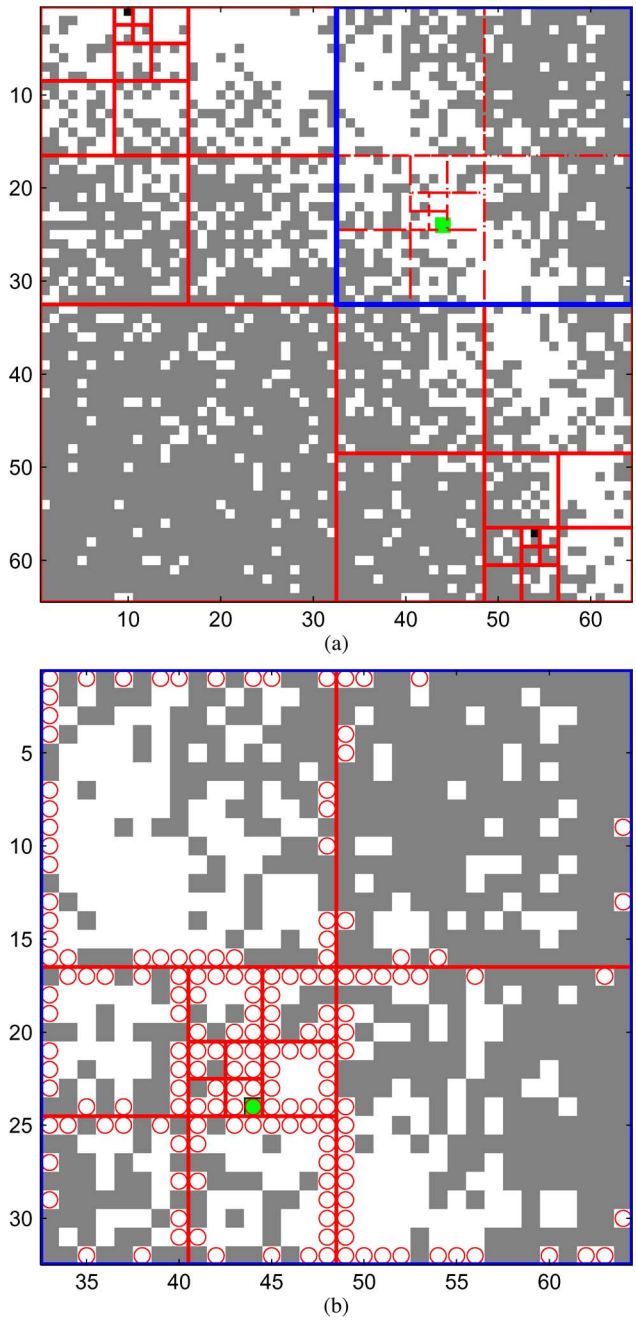


Fig. 3. (a) Illustration of the DPFR-RDP. The red grid shows the original PFR-RDP before any vertex update; the dashed red grid stands for the further partitioning after the green cell has been updated. The blue frame encloses the candidate d-square that is selected for further dyadic partitioning. (b) Magnified upper-right candidate d-square in (a). The free boundary cells are indicated by red circles. Except for one of the smallest newly-generated d-squares, all other d-squares contain the same inter-distance information as before, which is obtained through the bottom-up fusion process.

Algorithm 1 DPFR-RDP (dynamic path-finding reduced recursive dyadic partition)

1: Set the largest scale to $J = \log_2 n$, where the image size is n by n.

2: Initialize the list $dptree = [1, 1, 1]$–the d-square at the coarsest level.
3: **for** $s = 1 : J - 1$ **do**
4: For d-square at scale s in $dptree$
5: **if** v_s (source) or v_e (destination) is in this d-square **then**
6: In $dptree$, remove the line corresponding to this d-square;
7: Partition into four equal, smaller d-squares, and insert them as new lines in $dptree$
8: **end if**
9: **end for**
10: **if** v' (update) $\neq \varnothing$ **then**
11: Locate the d-square in PFR-RDP where v' locates, denoted as $[s_{v'}, a_{v'}, b_{v'}]$
12: Conduct PFR-RDP (which is the first **for** loop above) on $[s_{v'}, a_{v'}, b_{v'}]$ by setting $v_s = v_e = v'$
13: **end if**
14: **return** $dptree$

B. Update of Multiscale Information in the Beamlet Graph

In order to update the edge weights in the beamlet graph that were influenced by the updated cell, it would be far more redundant if we were to recalculate from scratch the inter-distances between all free boundary cells. In fact, this information has already been obtained during the bottom-up fusion process when we run m-A* at the initialization step (later on we show that LPA* runs exactly the same way as m-A* before the updates).

By taking advantage of the hierarchical inter-distance structure via the bottom-up fusion algorithm, the edge weights in the beamlet graph can be updated promptly. For instance, in Fig. 3(b), except for the smallest d-square where the green cell is located, no change of edge weights happens in any other d-square. Only the updates of the edge weights in the smallest d-square that contains the updated cell need to be recalculated, which is trivial, given the fact that the finest scale d-square contains only four cells. The main effort thus involves running an all-shortest path algorithm on the graph constructed from all the free boundary cells of the newly added d-squares during the further partitioning. More generally, multiple updates at the same time can be processed the same way. Fig. 4 shows a typical example of multiple updates performed simultaneously.

Algorithm 2 BottomUpFusion (For each d-square)

1: Read the parameters of each d-square: s (scale), a, b (location);
2: **if** $s = \log_2 n - 1$ **then**
3: Compute the free boundary cells as vertices (Trivial case: only four cells in the d-square)
4: Calculate the four nearest neighbor connectivity (edges) within each d-square
5: Run Johnson's algorithm on the resulting graph to obtain all pairs of shortest paths: $cgraph$ and $pathList$.
6: **end if**
7: **if** $s > 1$ **then**

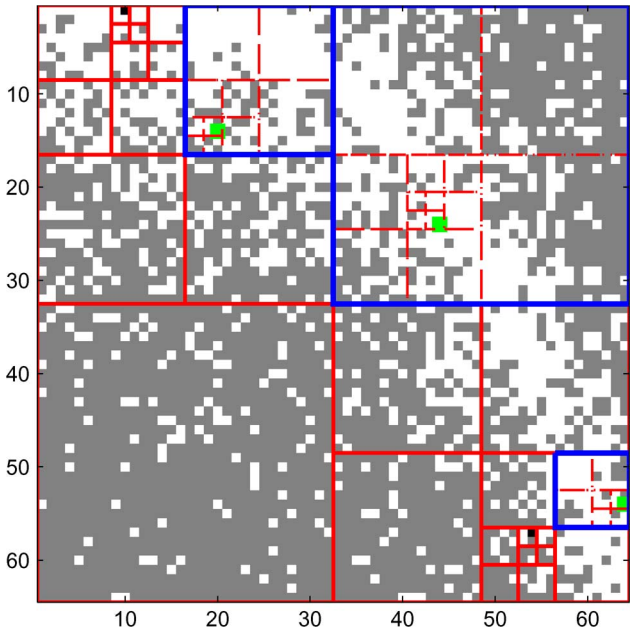


Fig. 4. Illustration of processing multiple updates simultaneously. The red grid shows the original PFR-RDP obtained from m-A*; the dashed red grid stands for the further partitioning after the green cells have been updated. The blue frame encloses the candidate d-squares marked for further partitioning.

8: $[graph1, path1] =$ BottomUpFusion$(s+1, 2a-1, 2b-1)$
9: $[graph2, path2] =$ BottomUpFusion$(s+1, 2a, 2b-1)$
10: $[graph3, path3] =$ BottomUpFusion$(s+1, 2a-1, 2b)$
11: $[graph4, path4] =$ BottomUpFusion$(s+1, 2a, 2b)$
12: Merge $graph1, \ldots, graph4$ into $Graph$ by adding the connected edges between neighboring d-squares
13: Run Johnson's algorithm on $Graph$ and get $cgraph$ and $tmpPathList$
14: Insert the missing parts of paths in $tmpPathList$ from $path1, \ldots, path4$ to obtain $pathList$
15: **return** $cgraph$, $pathlist$ (i.e., the beamlet graph)
16: **end if**

C. LPA* Algorithm on the Beamlet Graph

Now that we have established a method for updating the beamlet graph, we turn our attention to finding the shortest path on this graph. Given the source and destination, the first step of m-LPA* is to run m-A*. This gives us the beamlet graph and the initial optimal path. If there is no change in the environment, the algorithm terminates. When any cell is updated by some event, the dynamic PFR-RDP is constructed, and the multiscale inter-distance information obtained from the bottom-up fusion algorithm during the first step is used to update the vertices and edge weights in the beamlet graph. The final step is to run the LPA* algorithm on the updated beamlet graph. The whole algorithm is summarized in Algorithm 3.

Algorithm 3 Multiscale Lifelong A* (m-LPA*)

1: **procedure** $Key(v)$

2: **return** $[g(v) \wedge rhs(v) + h(v_s, v); g(v) \wedge rhs(v)]$
3: **procedure** Initialize()
4: $OPEN = \emptyset$;
5: for all $v \in V$ $rhs(v) = g(v) = \infty$;
6: $rhs(v_s) = 0$;
7: insert v_s with $Key(v_s)$ into $OPEN$
8: **procedure** UpdateState(v)
9: **if** $v \neq v_s$ **then**
10: $rhs(v) = \min_{v' \in Pred(v)}(c(v, v') + g(v'))$
11: **end if**
12: **if** $v \in OPEN$ **then**
13: remove v from $OPEN$
14: **end if**
15: **if** $g(v) \neq rhs(v)$ **then**
16: insert v into $OPEN$ with $Key(v)$
17: **end if**
18: **procedure** ComputeShortestPath()
19: **while** $\min_{v \in OPEN}(key(v)) < key(v_{\text{goal}}) | rhs(v_{\text{goal}}) \neq g(v_{\text{goal}})$ **do**
20: remove state v with min key from $OPEN$;
21: **if** $g(v) > rhs(v)$ **then**
22: $g(v) = rhs(v)$;
23: for all $v' \in Succ(v)$ UpdateState(v');
24: **else**
25: $g(v) = \infty$;
26: for all $v' \in Succ(v) \cup \{v\}$ UpdateState(v');
27: **end if**
28: **end while**
29: **procedure** Main()
30: Initialize img, v_s, v_e;
31: Conduct *PFR-RDP* and obtain $dptree$
32: Run the *Bottom-Up Fusion* algorithm on each d-square in $dptree$ and get beamlet graph
33: **for** ever **do**
34: ComputeShortestPath();
35: Wait for changes in edge costs;
36: quadtree = FurtherPartition();
37: Update beamlet graph via *Bottom-up Fusion*;
38: **for** all directed edges (u,w) with changed cost **do**
39: Update edge cost $c(u, w)$;
40: UpdateState(u);
41: **end for**
42: **end for**

V. Complexity Analysis and Data Structure

In this section we discuss the cost of replanning and the influence of the data structure used to maintain the priority queue on the overall computational complexity of the m-LPA* algorithm. Roughly speaking, complexity of any search algorithm depends on two factors: the number of vertex expansions, and the number of heap percolations. In Section V-A the worst-case scenario analysis of the algorithmic complexity in terms of vertex expansions is provided, lending support to the benefits of the proposed m-LPA* algorithm. Section V-B considers the implementation issues to reduce the heap percolation overhead.

A. Worst-Case Complexity Analysis

In order to investigate the complexity of the replanning step, and without loss of generality, we assume that only one vertex update occurs every single time. Let us denote the number of vertex expansions as V_e. We have the following theorem.

Theorem 1: In the worst case, $|V_e| = O(n^2)$ on the nearest neighbor graph, and $|V_e| = O(n)$ on the beamlet graph.

Proof: In the worst case, the complexity of replanning has the same order as the initial path-planning [1], and hence $|V_e^{\text{NNG}}| = O(n^2)$, where $|V_e^{\text{NNG}}|$ denotes the number of vertex expansions for the nearest neighbor graph.

In the replanning part, multiscale information has already been obtained via the bottom-up fusion algorithm. There are two scale-1, d-squares, and six scale-s, d-squares when $s \geq 2$. Furthermore, for a d-square at scale s, there are at most $n2^{2-s}$ free boundary pixels. Therefore, the initial beamlet graph has $|V_1| \leq 2 \times 2n + 6 \times n + 6 \times (n/2) + 6 \times (n/4) + \cdots = 4n + 6n + 3n + (3/2)n + \cdots \leq 16n$ vertices. Furthermore, the number of vertices added during the replanning step has an upper bound of $|V_2| \leq 3 \times n + 3 \times (n/2) + 3 \times (n/4) + \cdots = 6n$. This is because during further partitioning of the d-square where the update of a vertex takes place, we have three new d-squares at each scale (see Fig. 1 for an example). Hence, the total number of vertices during replanning is $|V| = |V_1| + |V_2|$, which is bounded by $|V| \leq 22n$. Thus, in the worst case, the number of vertex expansions in the replanning for the beamlet graph is of order $O(n)$. ■

B. Fibonacci versus Binomial Heap Implementation

An efficient data structure is required in order to maintain the priority queue and to find the minimum cost vertex at each step of the search algorithm with the least effort. A Fibonacci heap is used in m-LPA* to maintain the priority queue, instead of a binomial heap (used in [1]), because the Fibonacci heap has a better amortized running time than the binomial heap. During the search step of the LPA*, there are mainly three operations that change the content of the heap: 1) insert; 2) find-minimum; and 3) delete-minimum. The complexity of these operations is important because they are used to calculate the heap percolation, which is a metric for comparing the performance of algorithms. It is computed as the sum of the total number of operations that maintain the heap structure (i.e., number of swapped parent and child pairs in the heap). In a Fibonacci heap, the operations *insert* and *find-minimum* work in constant (i.e., O(1)) amortized time, while the operation *delete-minimum* works in $O(\log n)$ amortized time [23]. In a binomial heap, on the other hand, the complexity of these operations is the same for all, namely $O(\log n)$. Hence, the heap percolation for a binomial heap is $(n_i + n_d + n_f)O(\log n)$, whereas using a Fibonacci heap the heap percolation is $(n_i + n_f)O(1) + n_d O(\log n)$. Here, n_i, n_d and n_f denote number of insert, delete-minimum and find-minimum operations. This shows that the Fibonacci heap is a better choice for the data structure implementation in a replanning algorithm in terms of heap percolations.

Also, note that the core component of m-LPA* is the beamlet graph obtained from the preprocessed multiscale information,

which has a reduced number of vertices, but an increased number of insertion operations, due to the generalization of connectivity between non-adjacent cells. As a result, the operation complexity of m-LPA* is dominated by insertion operations, which is approximately of order $O(1)$, compared to that $O(\log n)$ of LPA*. This observation also justifies the advantage of m-LPA* over LPA* in terms of heap percolations.

VI. Simulation Studies

In this section we provide numerical experiments from several different scenarios, comparing the m-LPA* with the original LPA* algorithm. For each scenario, five randomly generated gridworlds are constructed, on which the comparison between m-LPA* and LPA* takes place. In the first scenario, we created an n-by-n image, assuming that the probability of a certain cell (indexed as (x, y), where $1 \leq x, y \leq n$) to be obstacle-free is given by $p(x, y) = \exp(-\gamma|\varphi(x, y)|)$, where the constant γ will be specified later. The intuition behind this model is that pixels (i.e., fine resolution cells) near the curve defined by $\varphi(x, y) = 0$ have higher probability to be free than the pixels far away from this curve.

In all numerical experiments, the total number of vertex expansions (V_e, the number of updates of the g-value of the vertices) is used as the metric to compare the efficiency of the two algorithms. We did not use heap percolation or CPU time, because CPU time is a machine-dependent metric, and heap percolation is approximately of order $O(1)$ per vertex expansion, as a direct result of using both a Fibonacci heap and a beamlet graph (see Section V-B above).

First, we compare the shortest paths and the number of expanded vertices during the initial planning step of LPA* and m-LPA*. Fig. 5 shows the results for one of the sample environments for the case when $\varphi(x, y) = y - x^2/n$. As seen in these plots, the number of expanded vertices during the initial planning step using the beamlet graph is much smaller than the one using the nearest neighbor graph. Multiscale information used in the initial step of planning significantly reduces the number of vertex expansions, and because of the use of Fibonacci heaps, the number of vertex expansions is the only step that is time-consuming.

Next, the number of vertex expansions during the replanning step of both LPA* and m-LPA* algorithms are compared. To this end, recall that the nearest neighbor graph and the beamlet graph are the underlined graphs in the LPA* and m-LPA* algorithms, respectively. Based on the intuition that when the updated vertex does not belong to the shortest path identified by either the LPA* or the m-LPA* algorithm, the number of vertex expansions tends to be small, in these examples we imposed the blocking vertex to belong in the set of vertices of the initial shortest path, except for the source and destination vertices.

For the scenario shown in Fig. 5, we conducted five experiments with randomly generated gridworlds. For each experiment, the updated vertices were on the initial shortest path and they were updated one-at-a-time sequentially. Fig. 6 shows the pattern of the number of vertex expansions for LPA* and m-LPA* for one of these numerical experiments, but the

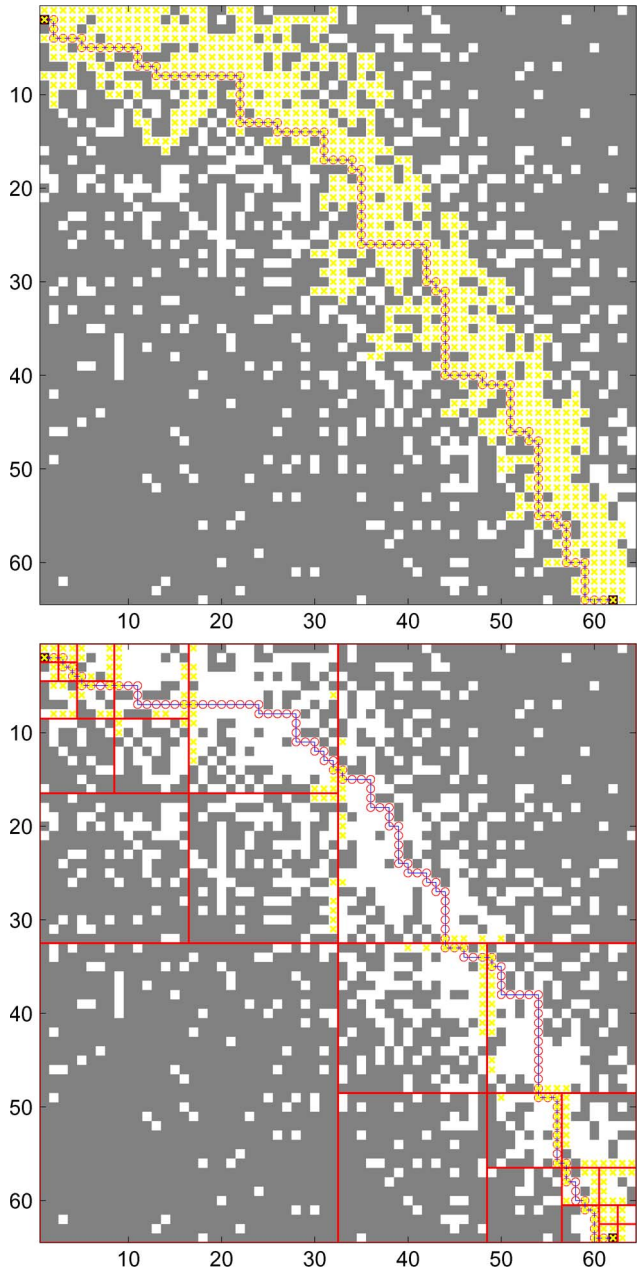


Fig. 5. (a) The shortest path identified by the first step of LPA* in the nearest neighbor graph. The black cells are the source and destination, whereas the gray cells stand for the obstacles. The red sequence of circles denotes the shortest path. The yellow crosses denote the expanded vertices in the initial planning. (b) The shortest path identified by the first step of m-LPA* in the beamlet graph.

pattern was consistent in all five experiments. The following observations are evident from Fig. 6:

1) The number of vertex expansions during replanning in LPA* varies dramatically from case to case. Specifically, when the updated vertex is closer to the source, or when the update generates a "local dead-end," a huge number of g-values may need to be recalculated.
2) The number of vertex expansions in m-LPA* is relatively insensitive with respect to the location of the updated vertices. The use of multiscale information reduces the number of vertex expansions when the blocking happens near the source; on the other hand, there will be a

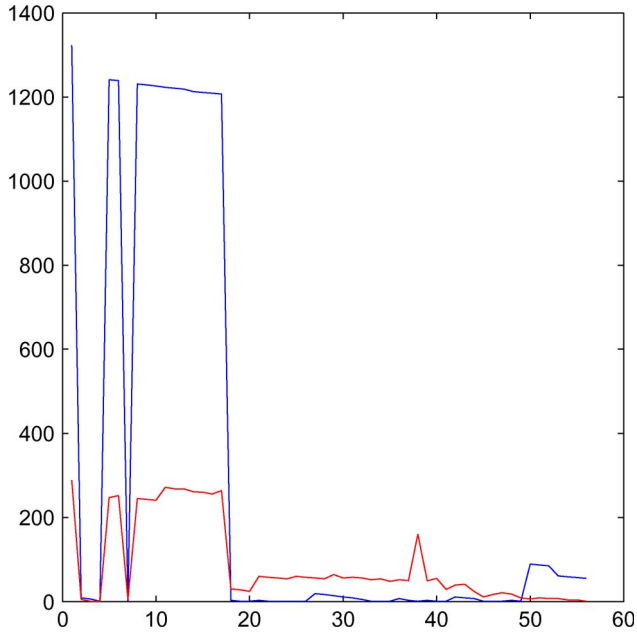


Fig. 6. Blue curve and the red curve denote the number of vertex expansions for LPA* and m-LPA* respectively. The number on the x-axis is the index of the updated vertex along the initial shortest path. For LPA*, the number of vertex expansions is highest when the updates occur close to the source, while for m-LPA*, the use of multiscale information alleviates the computations during replanning when the update is close to the source (with slightly increased computational burden when the update happens in the largest d-square); the net gain in terms of worst-case computational complexity is clear from this figure.

somewhat larger number of vertex expansions than that of LPA* when the update happens in the largest d-square in the dynamically recursive dyadic partition tree, because in this case more vertices will be added to the beamlet graph during the replanning step.

These observations are illustrated in greater detail in Figs. 7 and 8. Fig. 7 shows two cases of replanning obtained from the LPA* algorithm, which exhibited widely different numbers of vertex expansions. The black cells are the source and destination. The gray cells indicate obstacles. The solid blue dots denote the cells that changed their status and are not traversable. The red circles identify the original path during the first step of LPA* (essentially A*), and the green stars indicate the updated shortest path obtained from the replanning part of LPA*. The yellow crosses denote the expanded vertices during replanning. In Fig. 7(a), the blocking of a vertex in the gridworld induces a local "dead-end," and therefore all the g-values afterward are recalculated. The updated shortest path deviates a few steps before the location of the blocking cell, before converging back to the original path afterward. Because the blocking cell in Fig. 7(a) is much closer to the source than that in Fig. 7(b), the number of vertex expansions is also much higher, as expected.

Fig. 8 shows two cases of m-LPA* replanning. In Fig. 8(a) the blocking occurs near the source, as in Fig. 7(a), but the number of vertex expansions is much smaller, since the usage of multiscale information provides a smaller size graph, and hence a smaller number of vertex expansions as well. Fig. 8(b) illustrates the reason for an occasional increase in the number of vertex expansions in m-LPA*, especially when the vertex update occurs inside the largest d-square. In those cases, a larger number of free boundary cells is added to the beamlet graph as new vertices, following the recursive dyadic partitioning induced by the modified vertex. If the increased computational burden due to the newly-added vertices outperforms the gain from the multiscale structure of the beamlet graph, the number of vertex expansions can be higher than that of the LPA* implementation.

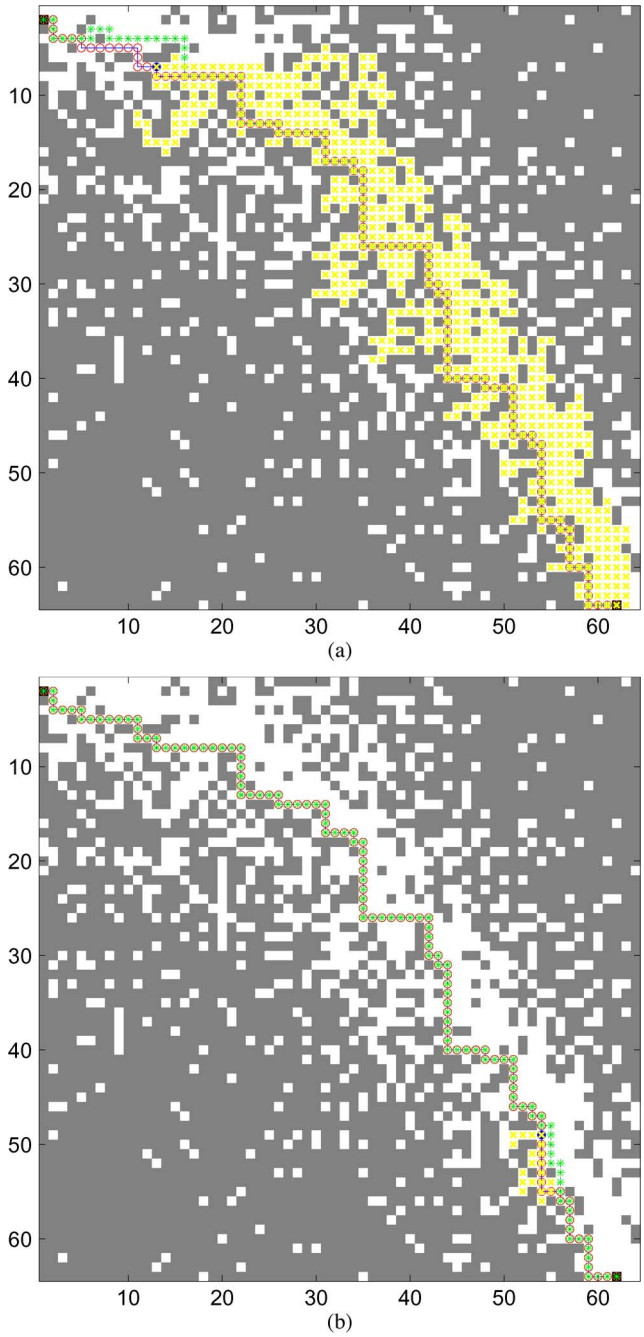


Fig. 7. (a) Shortest path obtained from LPA* in the nearest neighbor graph with a large number of vertex expansions during replanning. (b) Shortest path obtained from LPA* in the nearest neighbor graph with a small number of vertex expansions.

To further investigate the computational benefits obtained from the use of m-LPA*, we examined two more simulation scenarios as follows. We generated a gridworld similar to the previous case, but now we changed the parabolic curve to a circle. As before, comparisons were conducted using five

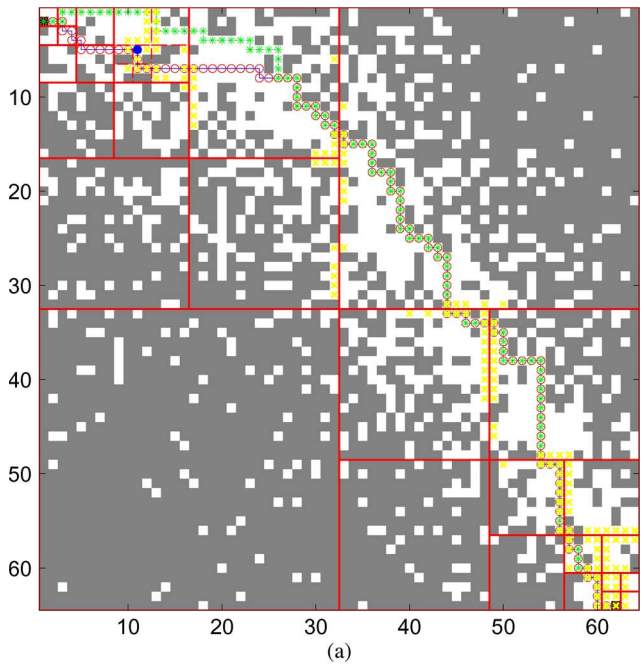


Fig. 8. Shortest path-planning obtained from m-LPA* in the beamlet graph.

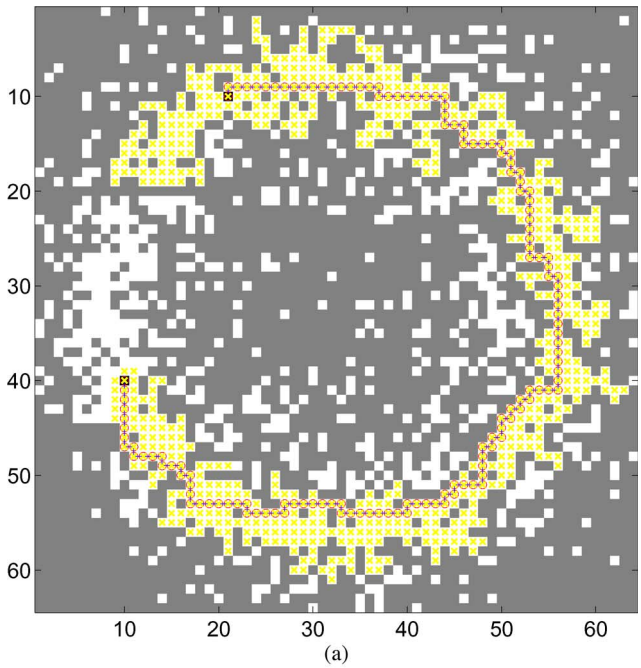


Fig. 9. (a) The shortest path obtained from LPA* in the nearest neighbor graph. Notice that all free cells in the nearest neighbor graph are expanded. (b) The updated shortest path obtained from m-LPA* in the beamlet graph. The use of m-LPA* results in a much smaller number of vertex expansions in the initial planning step.

randomly generated gridworlds. Fig. 9 shows the shortest paths obtained from the initial planning based on the nearest neighbor graph and the beamlet graph, respectively, while Fig. 10 shows two cases of replanning obtained from the LPA* and the m-LPA*, respectively.

The third scenario involves a sinusoidal curve embedded in the environment. Fig. 11 shows the initial planning obtained from the LPA* and the m-LPA*, and Fig. 12 shows two cases of replanning based on the nearest neighbor graph and the beamlet graph, respectively. Fig. 13 summarizes the results in terms of the number of vertex expansions for LPA* and m-LPA* for the last two scenarios. The number of vertex expansions for both LPA* and m-LPA* follows the same trend as in Fig. 6. For LPA*, the number of vertex expansions is highest when the update happens close to the source, while for m-LPA* this number varies according to the closeness of the vertex updates to the source, and whether the updates are located in the larger d-squares.

Finally, we reproduced a large-scale gridworld based on actual topographic data (i.e., elevation map) of a certain area in the US. Fig. 14 shows two cases of replanning based on the nearest neighbor graph and the beamlet graph, respectively. The difference in the number of expanded vertices for the two cases, is clearly shown in Fig. 14. Fig. 15(a) provides a

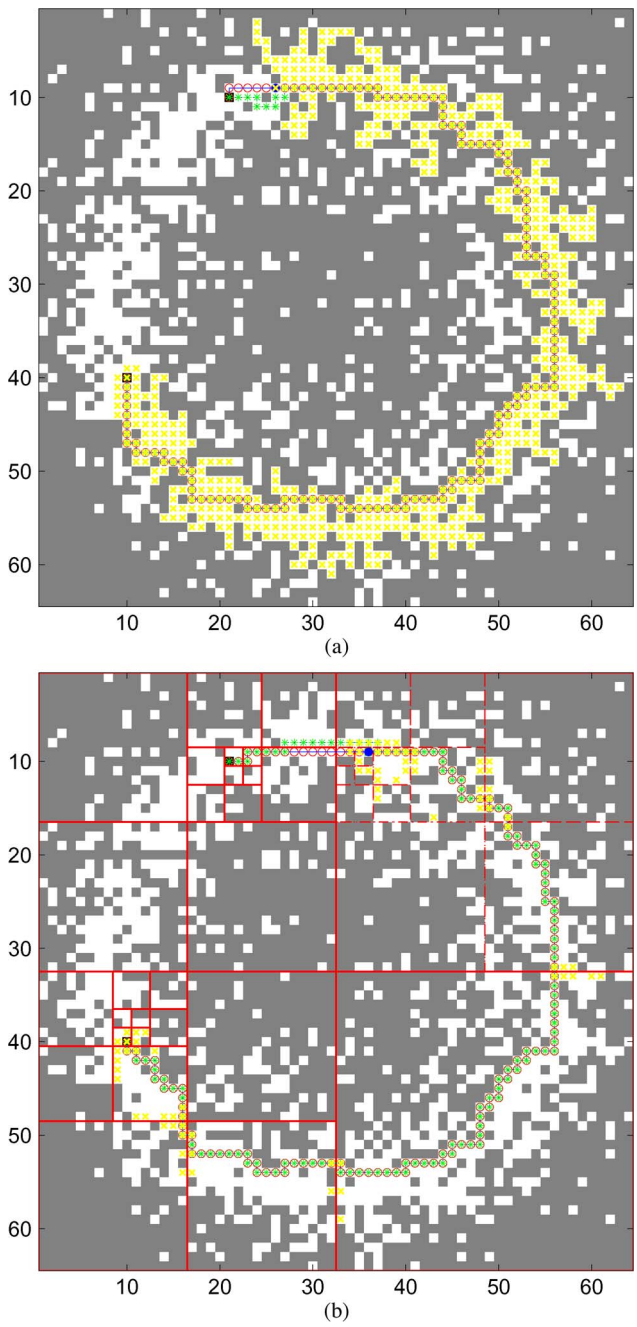


Fig. 10. (a) The updated shortest path obtained from LPA* in the nearest neighbor graph. The blocking happens near the source, and hence all the g-values afterward are recalculated. (b) The updated shortest path obtained from m-LPA* in the beamlet graph. The blocking happens in the upper-right coarsest scale d-square where further recursive dyadic partition takes place. Use of the multiscale information structure encoded in the DFPR-RDP significantly reduces the number of vertex expansions during replanning.

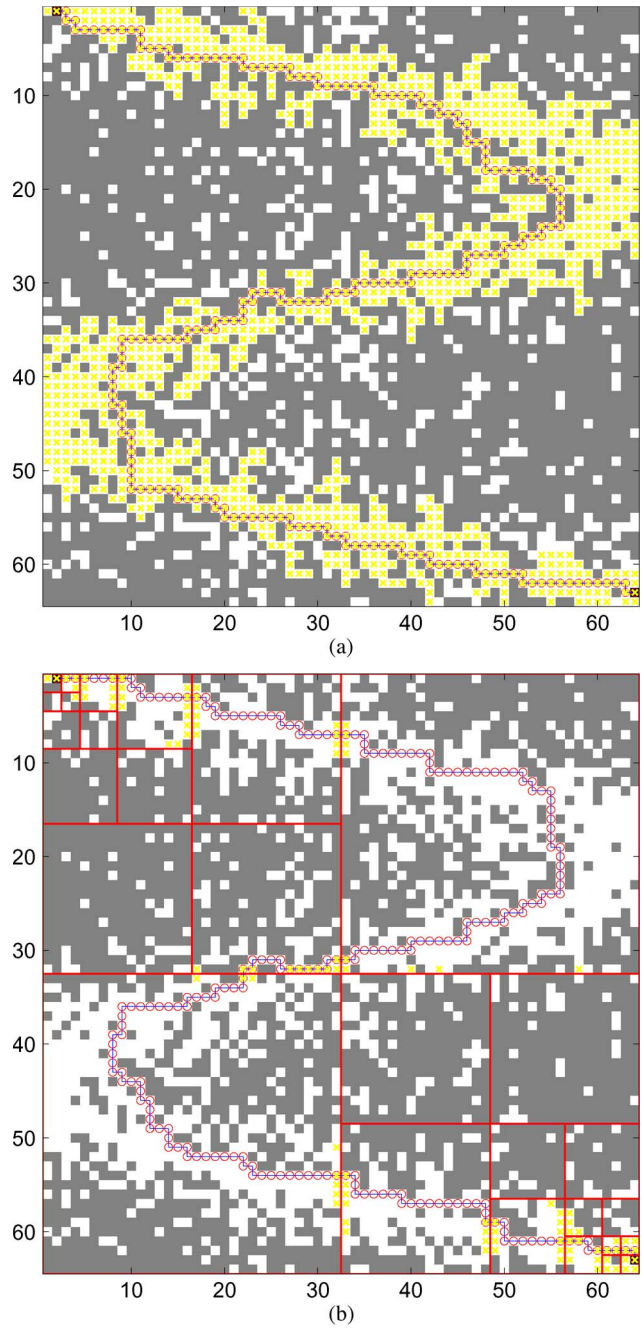


Fig. 11. (a) The initial shortest path obtained from the LPA* in the nearest neighbor graph. Notice that all free cells in the nearest neighbor graph are expanded in the initial planning. (b) The initial shortest path obtained from the m-LPA* in the beamlet graph. The use of multiscale information greatly reduces the number of vertex expansions during the initial planning step.

magnified version of Fig. 14(a) around the replanning area so that it can be seen clearly how the replanned path avoids the blocked vertex. Fig. 15(b) summarizes the results in terms of the number of vertex expansions for LPA* and m-LPA* for this example, which illustrates the effectiveness of our proposed m-LPA* algorithm for large scale, realistic maps.

The results of the numerical simulations in this section confirm that m-LPA* is a more robust algorithm than the LPA* in terms of the number of vertex expansions during replanning for a large variety of test scenarios. It is observed that the number of vertex expansions in m-LPA* does not vary dramatically with the location of the updated vertex whose traversability properties change. This mitigation of the volatility in the number of vertex expansions achieved by m-LPA* can be a desirable property in many applications.

VII. Discussion and Related Prior Work

The proposed m-LPA* algorithm provides a novel, nontrivial extension in the family of incremental search algorithms

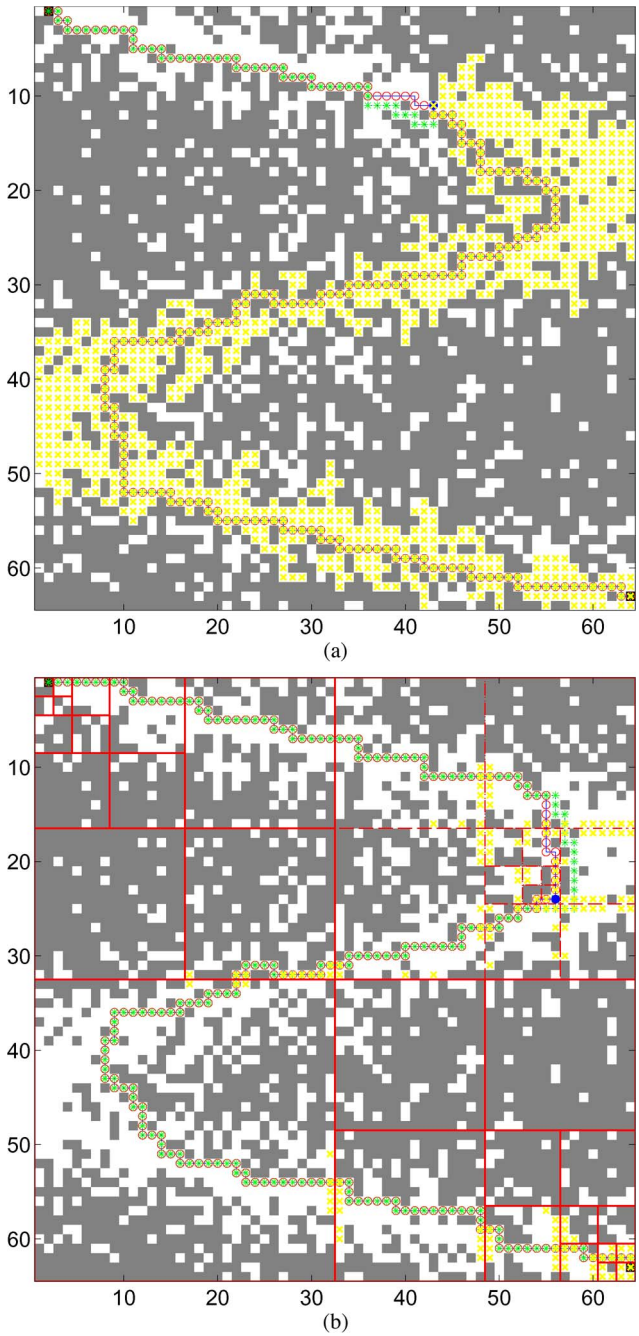


Fig. 12. (a) The updated shortest path obtained from LPA* in the nearest neighbor graph. The blocking happens near the source and hence all the g-values afterward are recalculated. (b) The updated shortest path obtained from m-LPA* in the beamlet graph. The blocking happens in the upper-right coarsest scale d-square where further recursive dyadic partition takes place. The use of multiscale information structure significantly reduces the number of vertex expansions during replanning.

operating on quadtree-based data structures. Quadtrees are, in fact, a widely adopted hierarchical representation used to encode obstacles in a given 2-D image/environment. They partition an image by recursively subdividing it into four smaller quadrants, and these successive subdivisions continue until either a subregion which is free of obstacles is found, or the finest resolution is reached. Because of their efficient memory requirements, quadtrees have become popular in path-planning applications.

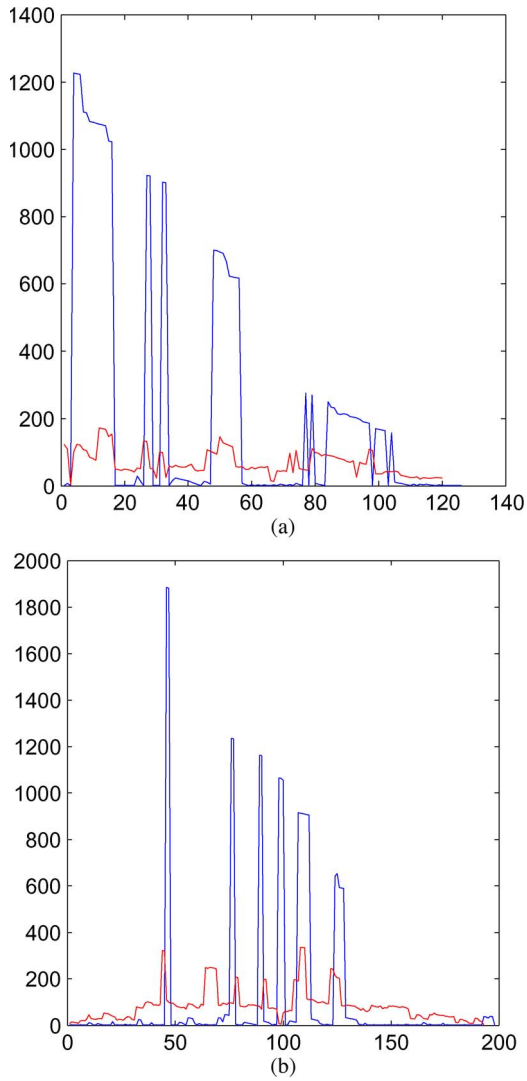


Fig. 13. Comparison of number of vertex expansions for both LPA* and m-LPA*. The blue curve and the red curve denote the number of vertex expansions for LPA* and m-LPA*, respectively. (a) Example environment of Figs. 9 and 10. (b) Example environment of Figs. 11 and 12.

Several previous path-planning algorithms exploit the nice properties of quadtrees. One of the earlier approaches is given in [24]. This work proved that a quadtree structure is better than the use of regular grids to represent 2-D environments. The benefits of quadtrees stem from the fact that they allow for an efficient partitioning of the environment so that single cells can be used to encode large empty regions. Quadtrees also reduce the memory requirements since they use a smaller number of cells. In early path-planning implementations, first a quadtree is created to represent the 2-D workspace, and then the path is generated by joining the line segments between the centers of the cells. However, this method merely finds a suboptimal solution. To remedy this shortcoming, the "framed quadtree" data structure was introduced in [25]. In a framed quadtree, free cells augmented the perimeter of each quadrant; these free cells are then used to pass through the different quadtree regions. The optimal path is generated by combining these free cells with line segments. At first glance, the framed quadtree data structure resembles the beamlet graph structure in the

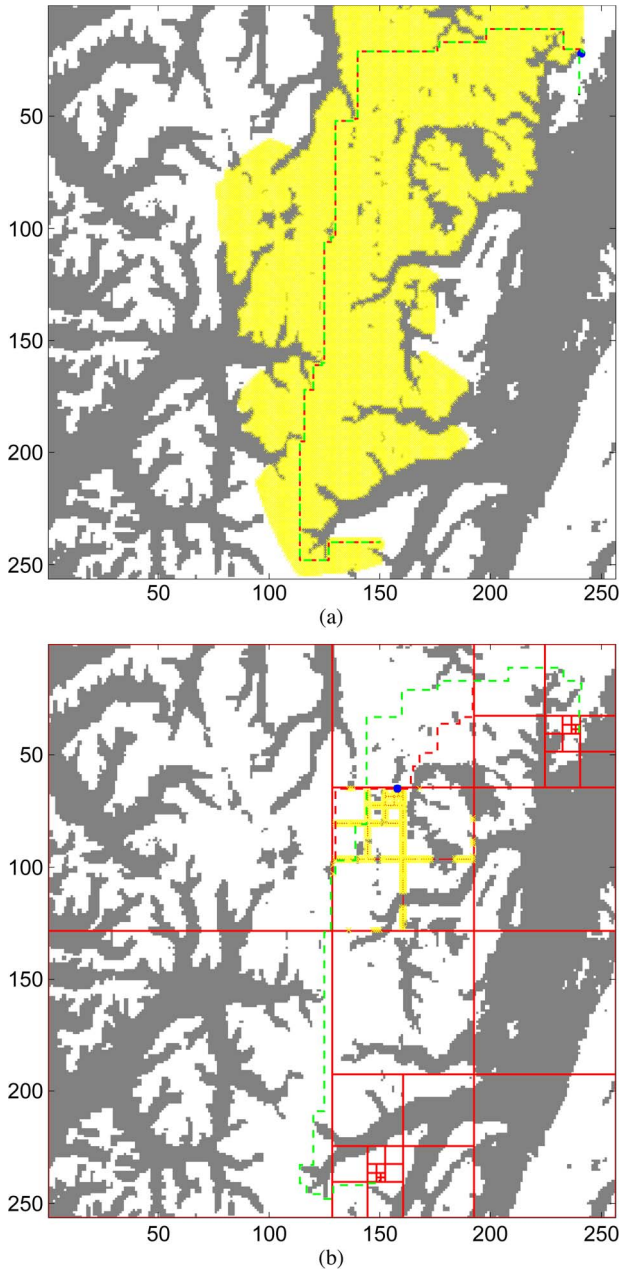


Fig. 14. Comparison between LPA* and m-LPA* for a large map with real topographic data. Gray pixels indicate obstacles and white pixels are free. (a) The blocking happens near the source and hence all the g-values afterward are recalculated. (b) The updated shortest path obtained from m-LPA* in the beamlet graph.

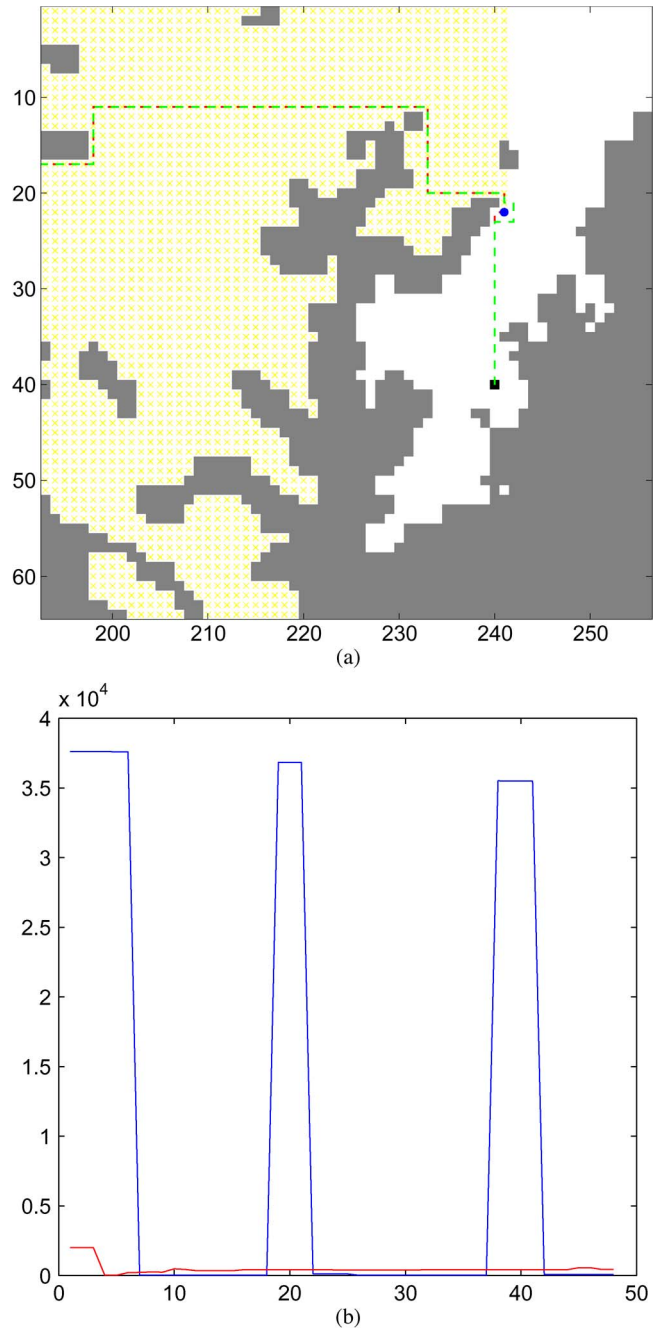


Fig. 15. (a) Magnified replanning area of the simulation shows the correctness of the m-LPA*. (b) Comparison of vertex expansions for both LPA* and m-LPA*. The blue curve and the red curve denote the number of vertex expansions for these two methods, respectively.

current paper. However, [25] does not introduce a beamlet-like connectivity to explore the finer scale information contained in the quadtree decomposition of the environment. Consequently, no "fusion" algorithm is used to efficiently organize the pre-computed information across all levels.

Another recent methodology that is somewhat similar to our work is the hierarchical path finding A* (HPA*) algorithm of [26]. This is a popular algorithm for real-time path-finding problems, especially for video game applications. The key idea of HPA* is to divide the map into clusters (corresponding to equally-sized squares) and generate a new graph (the so-called abstract graph) by using the information of the free cells belonging to the boundary of these clusters. The motivation behind such an approach is that for many real-time path-finding applications, the complete path is not needed. Knowing the first few moves of a valid path often suffices, allowing a mobile agent to start moving in the right direction, even before the whole path has been computed. Using this point of view, the authors of [26] derived a clustering algorithm that groups–when appropriate–neighboring nodes together to pre-process the grid and build a higher level graph (repeatedly if necessary) at multiple levels. The suboptimal path is found by searching at the top level first and then via recursively planning more detailed paths at the lower levels. Cached solutions of paths are

often reused in the hierarchy, so a tradeoff can be made between memory and computation. Enhancements of the original HPA* are provided in [27].

At first glance, HPA* bears a strong resemblance with m-LPA*, in the sense that they both construct a hierarchical decomposition of the environment. However, beamlet-like connectivity is absent in [26], [27] and the vertices of the abstract graph are just a subset of the original gridded map. Furthermore, at least two more significant differences exist compared with our work: first, the HPA* seeks an approximate solution fast, while the m-LPA* (and the m-A*) will find the exact, optimal solution to the shortest path problem. As mentioned above, HPA* was designed with computer games in mind, where only the initial few steps of the shortest path matters. This explains why the inexact solution is not a concern in HPA*. Instead, m-LPA* is designed when the exact shortest path needs to be found. Second, there are significant differences in the algorithmic design as well. For example, m-LPA* refines the gridworld into the finest resolution near the source and destination, whereas HPA* does not. Handling many scales in HPA* becomes difficult. The numerical examples in [26] use at most three levels. On the other hand, m-LPA* uses $\log n$ scales. The m-LPA* adopts a "bottom-up" fusion algorithm to organize pre-computed information. This is not present in HPA* either.

VIII. Conclusion

In this paper, we have presented a novel extension of the well-known Lifelong Planning A* (LPA*) algorithm based on a multiscale decomposition of the environment. Our algorithm may be viewed as an extension of both the classical LPA* algorithm and the recently proposed multiscale A* (m-A*) algorithm. The bottom-up fusion algorithm of m-A* is used as a multiscale strategy to preprocess the information at multiple scales and construct the beamlet graph. The proposed multiscale LPA* algorithm (m-LPA*) provides a significant reduction in terms of vertex expansions over the original LPA* algorithm at the worst case. Since the implementation of LPA* is based on a Fibonacci heap data structure for maintaining the priority queue, the vertex re-expansion dominates computations, which means that a significant reduction in terms of vertex re-expansions will result in a significant reduction in terms of computational time. These insights are confirmed by our numerical examples. Extensions of the proposed multiscale algorithm that adapts to a moving source, as well as extensions to higher dimensions are possible, and are currently under investigation.

Acknowledgment

Financial support for the work presented in this paper has been provided by NSF through award no. CMMI-0856565, which is gratefully acknowledged.

References

[1] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artif. Intell.*, vol. 155, no. 1/2, pp. 93–146, May 2004.

[2] Y. Lu, X. Huo, and P. Tsiotras, "Beamlet-like data processing for accelerated path-planning using multiscale information of the environment," in *Proc. 49th IEEE Conf. Decision Control*, Atlanta, GA, Dec. 2010, pp. 3808–3813.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.

[4] S. Kambhampati and L. S. Davis, "Multiresolution path planning for mobile robots," *IEEE J. Robot. Autom.*, vol. RA-2, no. 3, pp. 135–145, Sep. 1986.

[5] J. Y. Hwang, J. S. Kim, S. S. Lim, and K. H. Park, "A fast path planning by path graph optimization," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 33, no. 1, pp. 121–129, Jan. 2003.

[6] S. Behnke, *Local Multiresolution Path Planning*, vol. 3020. Berlin, Germany: Springer-Verlag, 2004, ser. Lecture Notes in Computer Science, pp. 332–343.

[7] C.-T. Kim and J.-J. Lee, "Mobile robot navigation using multiresolution electrostatic potential field," in *Proc. 31st IEEE IECON*, 2005, pp. 1774–1778.

[8] B. Sinopoli, M. Micheli, G. Donato, and T. J. Koo, "Vision based navigation for an unmanned aerial vehicle," in *Proc. IEEE Conf. Robot. Autom.*, 2001, pp. 1757–1764.

[9] D. Pai and L.-M. Reissell, "Multiresolution rough terrain motion planning," *IEEE Trans. Robot. Autom.*, vol. 14, no. 1, pp. 19–33, Feb. 1998.

[10] P. Tsiotras and E. Bakolas, "A hierarchical on-line path-planning scheme using wavelets," in *Proc. Eur. Control Conf.*, Jul. 2–5, 2007, pp. 2806–2812.

[11] R. Cowlagi and P. Tsiotras, "Multiresolution path planning with wavelets: A local replanning approach," in *Proc. Amer. Control Conf.*, Seattle, WA, Jun. 1–13, 2008, pp. 1220–1225.

[12] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy, "Incremental heuristic search in artificial intelligence," *Artif. Intell. Mag.*, vol. 25, no. 2, pp. 99–112, 2004.

[13] A. Stentz, "Optimal and efficient path planning for unknown and dynamic environments," *Int. J. Robot. Autom.*, vol. 10, no. 3, pp. 89–100, 1995.

[14] S. Koenig and M. Likhachev, " D* lite," in *Proc. AAAI Conf. Artif. Intell. (AAAI)*, 2002, pp. 476–483.

[15] D. Ferguson, M. Likhachev, and T. Stentz, "A guide to heuristic-based path planning," in *Proc. Int. Workshop Plan. Under Uncertainty Auton. Syst., ICAPS*, Jun. 2005, pp. 9–18.

[16] D. Ferguson and T. Stentz, "The field D* algorithm for improved path planning and replanning in uniform and non-uniform cost environments," Robot. Inst., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-RI-TR-05-19, Jun. 2005.

[17] A. Nash, K. Daniel, S. Koenig, and A. Felner, " Theta*: Any-angle path planning on grids," in *Proc. AAAI Conf. Artif. Intell. (AAAI)*, 2007, pp. 1177–1183.

[18] R. Cowlagi and P. Tsiotras, "Shortest distance problems in graphs using history-dependent transition costs with application to kinodynamic path planning," in *Proc. Amer. Control Conf.*, St. Louis, MO, Jun. 10–12, 2009, pp. 414–419.

[19] D. Donoho and X. Huo, "Beamlets and multiscale image analysis," in *Multiscale and Multiresolution Methods*, vol. 20. Berlin, Germany: Springer-Verlag, Spring 2002, pp. 149–196.

[20] D. Donoho and X. Huo, "Beamlets pyramids: A new form of multiresolution analysis, suited for extracting lines, curves and objects from very noise image data," *Proc. SPIE*, vol. 4119, no. 1, pp. 434–444, Jul. 2000.

[21] D. Donoho and X. Huo, "Applications of beamlets to detection and extraction of lines, curves, and objects in very noisy images," in *Proc. Nonlinear Signal Image Process.*, Jun. 2001.

[22] D. Donoho and X. Huo, "Beamlab and reproducible research," *Int. J. Wavelets, Multiresolution Inf. Process.*, vol. 2, no. 4, pp. 391–414, 2004.

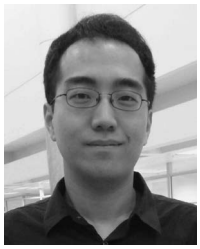
[23] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM (JACM)*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[24] H. Samet, "Neighbor finding techniques for image represented by quadtrees," *Comput. Graph. Image Process.*, vol. 18, no. 1, pp. 37–57, Jan. 1982.

[25] A. Yahja, A. Stentz, S. Singh, and B. L. Brumit, "Framed-quadtree path planning for mobile robots operating in sparse environments," in *Proc. IEEE Int. Conf. Robot. Autom.*, Leuven, Belgium, May 1998, pp. 650–655.

[26] A. Botea, M. Muller, and J. Schaeffer, "Near-optimal hierarchical pathfinding," *J. Game Develop.*, vol. 1, no. 1, pp. 7–28, 2004.

[27] M. R. Jansen and M. Buro, " HPA* enhancements," in *Proc. 3rd Artif. Intell. Interactive Digital Entertainment Conf.*, Jun. 2007, pp. 84–87.



Yibiao Lu received the B.S. degree in applied mathematics from the University of Science and Technology of China, Hefei, China, in 2008. He is currently working toward the Ph.D. degree at the School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta.

His research interests are applied statistics and machine learning.

Xiaoming Huo (S'96–M'99–SM'04) received the B.S. degree in mathematics from the University of Science and Technology, Hefei, China, in 1993, and the M.S. degree in electrical engineering and the Ph.D. degree in statistics from Stanford University, Stanford, CA, in 1997 and 1999, respectively.

Since August 2006, he has been an Associate Professor with the School of Industrial and Systems-break Engineering, Georgia Institute of Technology, Atlanta. He represented China in the 30th International Mathematical Olympiad (IMO), which was held in Braunschweig, Germany, in 1989, and received a golden prize. His research interests include statistics and multiscale methodology. He has made numerous contributions on topics such as sparse representation, wavelets, and statistical problems in detectability.

Dr. Huo was a Fellow of IPAM in September 2004. He won the Georgia Tech Sigma Xi Young Faculty Award in 2005. His work has led to an interview by Emerging Research Fronts in June 2006 in the field of Mathematics—every two months, one paper is selected.

Oktay Arslan received the B.S. degrees in control engineering and computer engineering from the Istanbul Technical University, Istanbul, Turkey, in 2006 and 2007, respectively. He also received the M.S. degree in defense technologies from the same institution in 2009. He is currently a Ph.D. student in the Department of Aerospace Engineering at Georgia Institute of Technology, Atlanta.

His current research interests include motion planning and task scheduling in command and control applications.

Panagiotis Tsiotras (S'90–M'92–SM'02) received the diploma in mechanical engineering from the National Technical University of Athens, Greece, in 1986, the M.S. degree in aerospace engineering from Virginia Tech, Blacksburg, in 1987, and the M.S. degree in mathematics and the Ph.D. degree in aeronautics and astronautics from Purdue University, West Lafayette, IN, in 1992 and 1993, respectively.

He is a Professor in the School of Aerospace Engineering at the Georgia Institute of Technology, Atlanta.

Dr. Tsiotras is a recipient of the NSF CAREER Award. His research interests are in optimal and nonlinear control and vehicle autonomy. He is a Fellow of AIAA.