2011 IEEE/RSJ International Conference on
Intelligent Robots and Systems
September 25-30, 2011. San Francisco, CA, USA

Multi-Scale LPA* with Low Worst-Case Complexity Guarantees

Yibiao Lu, Xiaoming Huo, Oktay Arslan, and Panagiotis Tsiotras

Abstract— **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 incremental algorithms exist in the literature that solve this problem, notably among them the Lifelong Planning A* (LPA*) algorithm. Although, in most cases, the LPA* algorithm 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.**

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 a well-known algorithm, widely used to solve dynamic path-planning problems, especially in mobile robotic applications. In numerical experiments it was observed however that LPA* exhibits unfavorable worst case performance. That is, the number of vertex updates can vary widely, depending on the location of the updated vertex in the graph. 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-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 m-LPA* algorithm 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 the path finding reduced recursive dyadic partitioning, or PFR-RDP for short). 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. 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.

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, [2], [3], [4], [5], [6] and, more recently, Refs. [7] and [8], [9] where wavelets are used to create several levels of abstraction for the environment. The paper also uses ideas similar to those appeared in [10], [11], where boundary cells in a cell decomposition as used to encode information about the environment and also on ideas of state and map abstractions [12], [13], [14], [15], whose aim is to repeatedly cluster information about the environment in a hierarchy of coarsening levels in order to enable fast post-processing. However, the paper takes the current state-of-the-art one step further by providing a systematic way to store and process information at multiple levels via a "bottom-up" recursive fusion algorithm. It should be noted that incremental heuristic search algorithms have been used extensively in many real-world applications, particularly in robotics. Map abstraction techniques, on the other hand, are currently widely utilized in the computer game industry. However, there is only a limited number publications that combine the principles of incremental search with map abstraction to achieve better runtime efficiency. In that respect, the paper aims at bridging a research gap in incremental search and map abstraction techniques.

II. Problem Formulation

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. As usual, it is assumed that the image contains two types of pixels: gray pixels (representing non-traversable obstacles) and 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].

Y. Lu is a Ph.D. candidate in the H. Milton Stewart School of Industrial and System Engineering, Georgia Institute of Technology, Atlanta, GA, USA, Email: ylv3@gatech.edu

X. Huo is with the faculty of the H. Milton Stewart School of Industrial and System Engineering, Georgia Institute of Technology, Atlanta, GA, USA, Email: huo@gatech.edu

O. Arslan is a Ph.D. candidate in the D. Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA, Email: oktay@gatech.edu

P. Tsiotras is with the faculty of D. Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA, Email: tsiotras@gatech.edu

[1] A cell is defined to be a collection of pixels in the environment. In this article, since we consider the highest resolution, cell and pixels are equivalent.

978-1-61284-456-5/11/$26.00 ©2011 IEEE

Among the several existing replanning algorithms [1], [16], [17], [18], [19], [20], [21], [22], in this paper we focus on LPA*, one of the most efficient replanning methods. The LPA* is an incremental search algorithm that replans the initial path when a vertex update occurs owing to a change in the environment. 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.

III. The Multiscale A* and Lifelong Planning A* Algorithms

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

The classical A* algorithm searches through all free cells in the environment, which can be overwhelmingly redundant. The motivation behind 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, summarized below. For a more in-depth description of m-A*, see [23].

(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 on the boundaries of each d-square in the PFR-RDP are selected as the vertices in the search graph. Figure 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 4-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 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 [24]. 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 [25], [26], and [27] 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. 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. The pseudo-code of the bottom-up fusion algorithm is given in Algorithm 1.

Algorithm 1 BottomUpFusion (For each d-square)

Read the parameters of each d-square: s (scale), a, b (location);
if $s = \log_2 n - 1$ **then**
  Compute the free boundary cells as vertices (Trivial case: only four cells in the d-square)
  Calculate the four nearest neighbor connectivity (edges) within each d-square
  Run Johnson's algorithm on the resulting graph to obtain all pairs of shortest paths: *cgraph* and *pathList*.
end if
if $s > 1$ **then**
  [*graph1, path1*] = BottomUpFusion($s+1, 2a-1, 2b-1$)
  [*graph2, path2*] = BottomUpFusion($s+1, 2a, 2b-1$)
  [*graph3, path3*] = BottomUpFusion($s+1, 2a-1, 2b$)
  [*graph4, path4*] = BottomUpFusion($s+1, 2a, 2b$)
  Merge *graph1, ..., graph4* into *Graph* by adding the connected edges between neighboring d-squares
  Run Johnson's algorithm on *Graph* and get *cgraph* and *tmpPathList*
  Insert the missing parts of paths in *tmpPathList* from *path1,..., path4* to obtain *pathList*
  return *cgraph, pathlist* (i.e., the beamlet graph)
end if

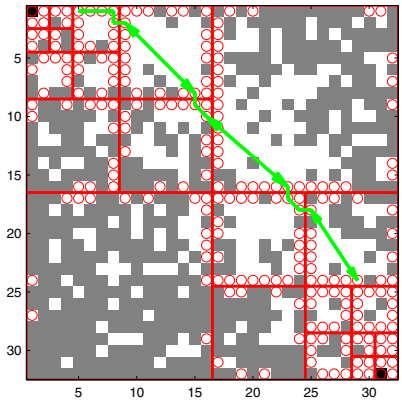


Fig. 1. Illustration of the Path-Finding Reduced Recursive Dyadic Partition (PFR-RDP) on a 32×32 image. The black cells are the source and destination. The red circles denote the free boundary cells, i.e., the vertices in the beamlet graph. The green arrows show the edge weights between two free cells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

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 [23]. 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 4-nearest-neighbor graph.

B. Incremental Search Algorithm: LPA*

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) and 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.

The LPA* algorithm uses two estimates of the start distance, namely, $g(v)$ and $rhs(v)$. 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. 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. 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}. 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*

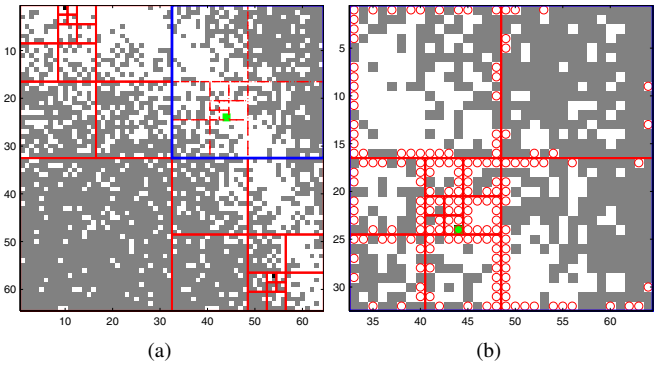
A. Dynamic Path-Finding Reduced Recursive Dyadic Partition

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. Figure 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 then we conduct a further partial dyadic partition (only) in this candidate d-square. Figure 2 shows the DPFR-RDP for a 64-by-64 image. The pseudo code for the DPFR-RDP is given in Algorithm 2.

Algorithm 2 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** $s = 2 : J - 1$ **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 on $[s_{v'}, a_{v'}, b_{v'}]$ by setting v_s= v_e= v'
13: **end if**
14: **return** *dptree*



(a) (b)

Fig. 2. (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.

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).

Thus, 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. 2(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.

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*)

procedure $Key(v)$
return $[g(v) \wedge rhs(v) + h(v_s, v); g(v) \wedge rhs(v)]$

procedure Initialize()
OPEN = $\emptyset$;
for all $v \in V$ $rhs(v) = g(v) = \infty$;
$rhs(v_s)$=0;
insert v_s with $Key(v_s)$ into *OPEN*

procedure UpdateState(v)
if $v \neq v_s$ **then**
  $rhs(v) = \min_{v' \in Pred(v)}(c(v, v') + g(v'))$
end if
if $v \in$ *OPEN* **then**
  remove v from *OPEN*
end if
if $g(v) \neq rhs(v)$ **then**
  insert v into *OPEN* with $Key(v)$
end if

procedure ComputeShortestPath()
while $\min_{v \in OPEN}(key(v)) < key(v_{\text{goal}}) || rhs(v_{\text{goal}}) \neq g(v_{\text{goal}})$ **do**
  remove state v with min key from *OPEN*;
  if $g(v) > rhs(v)$ **then**
    $g(v) = rhs(v)$;
    for all $v' \in Succ(v)$ UpdateState(v');
  else
    $g(v) = \infty$;
    for all $v' \in Succ(v) \cup \{v\}$ UpdateState(v');
  end if
end while

procedure Main()
Initialize *img*, v_s, v_e;
Conduct *PFR-RDP* and obtain *dptree*
Run the *Bottom-Up Fusion* algorithm on each d-square in *dptree* and get beamlet graph
for ever **do**
  ComputeShortestPath();
  Wait for changes in edge costs;
  quadtree = FurtherPartition();
  Update beamlet graph via *Bottom-up Fusion*;
  for all directed edges (u,w) with changed cost **do**
    Update edge cost $c(u, w)$;
    UpdateState(u);
  end for
end for

V. 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.

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 $|V_1| \leq 2 \times 2n + 6 \times n + 6 \times \frac{n}{2} + 6 \times \frac{n}{4} + ... = 4n + 6n + 3n + \frac{3}{2}n + ... \leq 16n$ vertices. Furthermore, the number of vertices added during the replanning step has an upper bound of $|V_2| \leq 3 \times n + 3 \times \frac{n}{2} + 3 \times \frac{n}{4} + \cdots = 6n$. 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 $O(n)$. □

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. In particular, 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 complexity of m-LPA* is dominated by insertion operations, which is approximately of order $O(1)$, compared to that $O(\log n)$ of LPA*, when a Fibonacci heap is used [28]. Hence, the overall complexity of the algorithm is dominated by the number of vertex expansions, hence it is of order $O(n)$.

VI. Simulation Studies

Several numerical examples were conducted to validate the performance of m-LPA* and compare it to the performance of LPA*. Owing to page limitations, in this section we provide the results from the numerical experiments from one of the scenarios tested, but the results are representative for all other cases. A large-scale gridworld based on actual topographic data (i.e., an elevation map) of a certain area in the US, as shown in Fig. 3, was used as the environment. The black cells in this figure are the source and destination. The gray cells indicate obstacles.

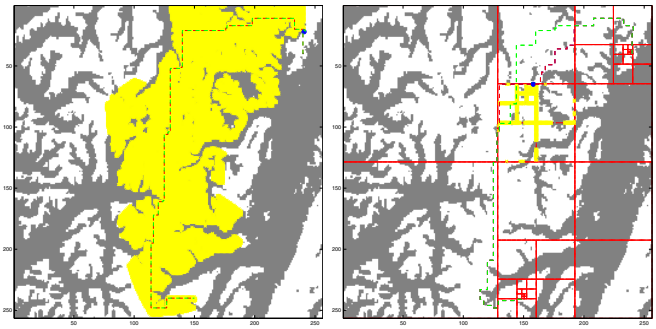
On this map, we run both the standard LPA* and the m-LPA* algorithms. The total number of vertex expansions (i.e., the number of updates of the g-value of the vertices) was used as the metric to compare the efficiency of the two algorithms. We did not use heap percolation or CPU time, because CPU time is a machine-dependent metric, and heap percolation is of order $O(1)$, as a direct result of using both a Fibonacci heap and a beamlet graph (see Section V above). 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, we imposed that the blocking vertex be from the set of vertices of the initial shortest path, except for the source and destination vertices. Hence, for each experiment, the updated vertices were on the initial shortest path and they were updated one-at-a-time sequentially.

The difference in the number of expanded vertices for the two cases, is clearly shown in Fig. 3. The solid blue dot denote the cell that changed its status and is not traversable in this particular case. The red dashed line identifies the original path during the first step of LPA* (essentially A*), and the green dashed line indicates the updated shortest path obtained from the replanning part of LPA*. The yellow crosses denote the expanded vertices during replanning. As shown in Fig. 3(b), the blocking of the vertex in the gridworld occurs near the start point and induces a local "dead-end." As a result, all the g-values afterwards are recalculated. Figure 4(a) provides a magnified version of Fig. 3(a) around the replanning area so that it can be seen clearly how the replanned path avoids the blocked vertex.

Figure 4(b) shows the pattern of the number of vertex expansions for LPA* and m-LPA*, respectively. As seen in this plot, 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.

The following observations are evident from Fig. 4(b):

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 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.



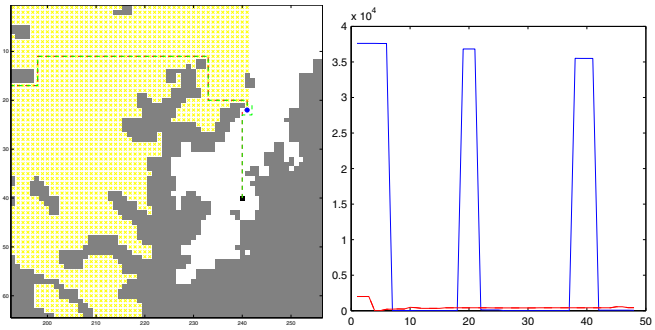
(a) (b)

Fig. 3. 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 afterwards are recalculated. (b) The updated shortest path obtained from m-LPA* in the beamlet graph.

As a general rule, 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.

VII. Conclusions

We have presented a new extension of the well-known Lifelong Planning A* (LPA*) algorithm based on a multiscale decomposition of the environment. The proposed algorithm may be viewed as an extension of both the classical



(a) Magnified Area in Fig. 3 (b) Example of Fig. 3

Fig. 4. (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.

LPA* algorithm and the recently proposed multiscale A* (m-A*) algorithm. The proposed multiscale LPA* algorithm (m-LPA*) provides a significant reduction in terms of vertex expansions over the original LPA* algorithm at worst case. 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.

REFERENCES

[1] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artificial Intelligence Journal*, vol. 155, no. 1-2, pp. 93–146, 2004.

[2] S. Kambhampati and L. S. Davis, "Multiresolution path planning for mobile robots," *IEEE Journal of Robotics and Automation*, vol. 2, no. 3, pp. 135–145, 1986.

[3] J. Y. Hwang, J. S. Kim, S. S. Lim, and K. H. Park, "A fast path planning by path graph optimization," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 33, no. 1, pp. 121–127, January 2003.

[4] S. Behnke, *Local Multiresolutin Path Planning*, ser. Lecture Notes in Computer Science. Berlin: Springer, 2004, vol. 3020, pp. 332–343.

[5] C.-T. Kim and J.-J. Lee, "Mobile robot navigation using multi-resolution electrostatic potential field," in *32^{nd} Annual Conference of IEEE Industrial Electronics Society, 2005, IECON 2005*, 2005.

[6] B. Sinopoli, M. Micheli, G. Donato, and T. J. Koo, "Vision based navigation for an unmanned aerial vehicle," in *Proceedings of 2001 IEEE Conference on Robotics and Automation*, 2001, pp. 1757–64.

[7] D. Pai and L.-M. Reissell, "Multiresolution rough terrain motion planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 19–33, 1998.

[8] P. Tsiotras and E. Bakolas, "A hierarchical on-line path-planning scheme using wavelets," *European Control Conference*, July 2-5 2007.

[9] R. Cowlagi and P. Tsiotras, "Multiresolution path planning with wavelets: A local replanning approach," in *American Control Conference*, Seattle, WA, June 1-13 2008, pp. 1220–1225.

[10] A. Yahja, A. Stentz, S. Singh, and B. L. Brumit, "Framed-quadtree path planning for mobile robots operating in sparse environments," in *Proceedings of the 1998 IEEE International Conference on Robotics & Automation*. Leuven, Belgium: IEEE, May 1998, pp. 650–655.

[11] M. Goldenberg, A. Felner, N. Sturtevant, and J. Schaeffer, "Portal-Based True-Distance Heuristics for Path Finding," in *Third Annual Symposium on Combinatorial Search*, 2010.

[12] A. Botea, M. Muller, and J. Schaeffer, "Near-optimal hierarchical pathfinding," *Journal of Game Development*, vol. 1, pp. 1–30, 2004.

[13] M. R. Jansen and M. Buro, "HPA* enhancements," in *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference*. Stanford, CA: The AAAI Press, June 2007, pp. 84–87.

[14] A. Felner, N. Sturtevant, and J. Schaeffer, "Abstraction-based heuristics with true distance computations," *Proceedings of the Eighth Symposium on Abstraction, Reformulatoin, and Approximation*, 2009.

[15] N. Sturtevant and M. Buro, "Partial pathfinding using map abstraction and refinement," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 20, no. 3, 2005, p. 1392.

[16] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy, "Incremental heuristic search in artificial intelligence," *Artificial Intelligence Magazine*, vol. 25, no. 2, pp. 99–112, 2004.

[17] S. Koenig and M. Likhachev, "D* lite," *Proceedings of the AAAI Conference of Artificial Intelligence (AAAI)*, pp. 476–483, 2002.

[18] A. Stentz, "Optimal and efficient path planning for unknown and dynamic environments," *International Journal of Robotics & Automation*, vol. 10, no. 3, pp. 89–100, 1995.

[19] D. Ferguson, M. Likhachev, and T. Stentz, "A guide to heuristic-based path planning," in *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[20] D. Ferguson and T. Stentz, "The field D* algorithm for improved path planning and replanning in uniform and non-uniform cost environments," Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-05-19, June 2005.

[21] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2007, pp. 1177–1183.

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

[23] Y. Lu, X. Huo, and P. Tsiotras, "Beamlet-like data processing for accelerated path-planning using multiscale information of the environment," in *The 49th IEEE Conference on Decision and Control*, Hilton Atlanta Hotel, Atlanta, GA, USA, Dec 2010.

[24] D. Donoho and X. Huo, *Multiscale and Multiresolution Methods*. Spring, 2002, vol. 20, ch. Beamlets and multiscale image analysis, pp. 149–196.

[25] ——, "Beamlets pyramids: A new form of multiresolution analysis, suited for extracting lines, curves and objects from very noise image data," in *Proceedings of SPIE*, vol. 4119, no. 1, July 2000, pp. 434–444.

[26] ——, "Applications of beamlets to detection and extraction of lines, curves, and objects in very noisy images," in *Proceedings of Nonlinear Signal and Image Processing*, June 2001.

[27] ——, "Beamlab and reproducible research," *International Journal of Wavelets, Multiresolution and Information Processing*, vol. 2, no. 4, pp. 391–414, 2004.

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