2016 IEEE International Conference on Robotics and Automation (ICRA)
Stockholm, Sweden, May 16-21, 2016

Reduced Complexity Multi-Scale Path-Planning on Probabilistic Maps

Florian Hauer[1] Panagiotis Tsiotras[2]

***Abstract*— We present several modifications to the previously proposed MSPP algorithm that can speed-up its execution considerably. The MSPP algorithm leverages a multi-scale representation of the environment in n dimensions encoded in tree structure constructed by recursive dyadic partitioning of the search space. We first present a new method to compute the graph neighbors in order to reduce the complexity of each iteration, from $O(|V|^2)$ to $O(|V|\log|V|)$. We then show how to delay expensive intermediate computations until we know that new information will be required, hence saving time by not operating on information that is never used during the search. Finally, we present a way to remove the very expensive need to calculate a full multi-scale map with the use of sampling and derive an upper bound on the probability of failure as a function of the number of samples.**

I. Introduction

Path-planning algorithms rely on perception algorithms to map the environment and localize the agent in the map. Commonly used data structures to represent perceived environments include multiresolution representations resulting from many common perception algorithms [2], [8]. The use of hierarchical multiresolution data structures is motivated by several observations: First, information collected about the environment is not uniform, as each sensor has its own range, resolution and noise properties. The information used by perception algorithms is then naturally multi-scale; estimation and inference is often used to extract the best information out of noisy measurements, leading to a probabilistic representation of the environment [8]. Second, on-board computational resources might be limited, thus not allowing an agent to systematically use all perceived information. Furthermore, precise information about the environment far away from the agent might not be valid, or may even be irrelevant, if the robot is far away from the obstacle. A multi-scale representation of the collected data allows to choose the resolution for each region of the space as needed. For planning purposes, for instance, local information is typically important over the short-term (e.g., obstacle avoidance), while far away information affects only long-term objectives such as reaching a goal or exploring the environment.

Several approaches have been used in the past to incorporate multiscale information during planning. Bottom-up approaches use the information at the finest resolution and then combine it in increasingly coarser resolutions. Top-down approaches solve the path-planning problem at the coarser resolution and then progressively increase the resolution of the solution [5], [7]. Using both approaches can lead to fast optimal algorithms, as shown in [6]. But the preprocessing of the data required by this approach is too expensive (in terms of time and memory) for online applications. Another approach consists of using the information at different resolutions at the same time. This idea is explored in [1], where areas near the current vehicle are represented accurately, while farther-away areas are coarsely-encoded by using a transformation on the wavelet coefficients. The approach is shown to be complete and very fast.

More recently, the MSPP algorithm [4] extended the work of [1] to n dimensions in a reformulation using 2^n trees instead of wavelets. The notion of ε-obstacles guaranteeing completeness for any value of the threshold ε was also introduced in the same paper. In this paper, we propose several modifications to the MSPP algorithm introduced in [4] to accelerate computations and extend the range of potential applications. Neighbor checking, a bottleneck operation of the MSPP algorithm, is reworked to reduce its complexity from $O(|V|^2)$ to $O(|V|\log|V|)$. The order in which operations are executed is also modified in order to minimize the computations. The full reduced graph is not computed, only its nodes are computed and the edges are calculated on-the-fly. Finally, we introduce a way to work without a multi-scale map, but instead we use a predicate to determine whether a point of the search space is an obstacle or not. This modification allows us to remove the expensive map creation step. It also saves memory, since the full multi-scale map does not need to be known in advance. The trade-off here is giving up completeness for the sake of increased runtime performance and memory reduction.

II. Notation and Previous Work

A. Multiresolution World Representation

The environment $\mathcal{W} \subset \mathbb{R}^d$ is assumed to be a d-dimensional grid world. Without loss of generality, we assume that each elementary cell of the grid world is a unit hypercube and there exists an integer $\ell > 0$ such that the world is contained within a hypercube of side length 2^ℓ. The world $\mathcal{W}$ is encoded as a tree $\mathcal{T} = (\mathcal{N}, \mathcal{R})$ representing the multi-scale information, with $\mathcal{N}$ being the set of nodes and $\mathcal{R}$ being the set of edges describing their relations. Nodes of $\mathcal{T}$ are represented by two indices, k and p, corresponding to the depth of the node $n_{k,p}$ in the tree and the position of the center of the node in the search space. A node $n_{k,p}$ represents a hypercube in the search space $\mathcal{W}$ centered at p and of size 2^k, and is denoted by $H(n_{k,p})$. The *children* of the node $n_{k,p}$ are denoted by $n_{k-1,q_i}, i \in [1, 2^d]$ where $q_i = p + 2^{k-2}e_i$ and where e_i is each of the 2^d (d-dimensional) vectors generated

[1]PhD candidate, School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0150, Email:fhauer3@gatech.edu

[2]Professor, School of Aerospace Engineering and Institute for Robotics and Intelligent Machines, Georgia Institute of Technology, Atlanta, GA 30332-0150, Email:tsiotras@gatech.edu

Support for this work has been provided by ARO MURI award W911NF-11-1-0046 and ONR award N00014-13-1-0563

by $[\pm 1, \pm 1, \ldots, \pm 1]$. A node is called a *leaf* of $\mathcal{T}$ if it has no children.

The information $V(n_{k,p})$ contained in each node $n_{k,p} \in \mathcal{N}$ is the probability of the existence of an obstacle in the space represented by the node, computed by

$$V(n_{k,p}) = \frac{\text{Volume of obstacles in } H(n_{k,p})}{\text{Volume of } H(n_{k,p})}. \quad (1)$$

B. The Path-Planning Problem

Two nodes in $\mathcal{T}$ are *neighbors* if their corresponding hypercubes share a hyperface, specifically, their intersection is a hypercube of dimension $d-1$. A necessary and sufficient condition for two nodes n_{k_1,p_1} and n_{k_2,p_2} to be neighbors is that both of the following two conditions are satisfied:

- The expression $\|p_1 - p_2\|_\infty = 2^{k_1-1} + 2^{k_2-1}$ holds,
- There exists a unique $i \in [1, d]$, such that $|(p_1 - p_2)_i| = 2^{k_1-1} + 2^{k_2-1}$, where $(p_1 - p_2)_i$ is the i^{th} component of the vector.

In the case of a 2-D or a 3-D environment with a uniform grid, the previous two conditions imply 4-connectivity or 6-connectivity, respectively.

We define a *path* $\pi = (n_{k_1,p_1}, n_{k_2,p_2}, \ldots, n_{k_N,p_N})$ in $\mathcal{T}$ to be a sequence of nodes $n_{k_i,p_i} \in \mathcal{N}$, each at corresponding position p_i and depth $\ell - k_i$, such that two consecutive nodes of the sequence are neighbors. A path is called a *finest information path* (FIP) if all its nodes are leafs of $\mathcal{T}$. Leaf nodes represent the best resolution, and hence the finer information contained in the tree $\mathcal{T}$.

Given $\varepsilon \in [0, 1)$, a node $n_{k,p} \in \mathcal{N}$ is an *ε-obstacle* if

$$V(n_{k,p}) \geq 1 - 2^{-dk}\varepsilon. \quad (2)$$

A path π is *ε-feasible* if none of its nodes are ε-obstacles.

Given the representation of $\mathcal{W}$ encoded in the tree $\mathcal{T}$, the problem is to find an ε-feasible FIP between two nodes in the tree, n_{start}, representing the starting node, and n_{goal}, representing the goal node, and to report failure if no such path exists.

C. The MSPP Algorithm

The MSPP algorithm is a backtracking algorithm that iteratively builds a solution from the starting node n_{start} until the goal node n_{goal} is reached. At each iteration i, a local representation $\mathcal{G}_i$ of the environment, called the *reduced graph*, is computed and the best path to the goal on this graph is used as a heuristic to decide which direction to follow. The current candidate solution built by the algorithm at iteration i is denoted by π^i_{start}. Therefore, π^i_{start} is an ε-feasible FIP from n_{start} to n_{k_i,p_i}, the node reached by the algorithm at iteration i. The best path from n_{k_i,p_i} to the goal on the reduced graph $\mathcal{G}_i$ is denoted by π^{goal}_i. The reduced graph $\mathcal{G}_i$ is computed by first identifying its vertices (corresponding to nodes of $\mathcal{T}$) via a top-down exploration of $\mathcal{T}$. The first element of π^{goal}_i is used to build the global solution π^i_{start}. If the path π^{goal}_i does not exist, the algorithm backtracks.

The main lines of the MSPP algorithm are shown in Algorithm 1. Refer to [4] for the full details of the MSPP algorithm.

Algorithm 1: The MSPP Algorithm - Simplified for clarity

Data: Tree $\mathcal{T}$, Start node n_{start}, Goal node n_{goal}
Result: ε-feasible FIP from n_{start} to n_{goal} or failure

```
i ← 0, n_{k_i,p_i} ← n_start, π^0_start ← [n_{k_i,p_i}];
while Goal not found AND no failure do
    (G̃_i, v_{start,i}, v_{goal,i}) ←ReducedGraph(T, n_{k_i,p_i});
    π^goal_i ←ShortestPath(G̃_i, v_{start,i}, v_{goal,i});
    if exists(π^goal_i) then
        n_{k_{i+1},p_{i+1}} ←firstElement(π^goal_i);
        π^{i+1}_start ← [π^i_start  n_{k_{i+1},p_{i+1}}];
    else
        backtrack;
    i ← i + 1;
```

III. Fast Neighbor Computation - MSPP-FN

In the original MSPP algorithm [4], neighbors for the reduced graph are computed by testing whether every pair of vertices satisfies the neighborhood properties. As shown in [4], this step is the main bottleneck during each iteration, having complexity $O(|V|^2)$, where $|V|$ is the number of vertices. In this section we propose a new way to compute neighbors so that the complexity is reduced from $O(|V|^2)$ to $O(|V| \log |V|)$.

A. Tree Data Structure For Vertices

In order to perform fast searches over the vertices of the reduced graph, we keep them in a tree structure. Let $\mathcal{T}_i$ define this tree structure. As the original tree $\mathcal{T}$ is traversed to select nodes for $\mathcal{G}_i$, $\mathcal{T}_i$ is constructed by copying every element of $\mathcal{T}$ traversed by the selection process, except for ε-obstacles. $\mathcal{T}_i$ is then a tree with the same structure as $\mathcal{T}$, but its branches are shorter. In other words, $\mathcal{T}_i$ is a pruned version of $\mathcal{T}$ whose leave nodes are the vertices of $\mathcal{G}_i$.

Note that, for implementation, $\mathcal{T}_i$ does not change significantly between two consecutive iterations, so it is computationally cheaper to modify $\mathcal{T}_{i-1}$ than to create a new data structure at each iteration. Memory allocation is the most expensive operation when creating new nodes. Modifying $\mathcal{T}_{i-1}$ allows to only have to allocate memory for nodes of $\mathcal{T}_i$ that did not exist in $\mathcal{T}_{i-1}$. The copy process is then modified to add nodes only if they do not already exist, and remove excessive nodes when reaching a node corresponding to a vertex of $\mathcal{G}_i$. The pseudo-code is given in Function 1. The vertex list is also removed since the information is already contained in $\mathcal{T}_i$. The function `GetRGFastNeighbor` is called with the root of $\mathcal{T}$, the root of $\mathcal{T}_i$ (created during initialization) and the current node n_{k_i,p_i}.

B. Same Size Neighbors

Generating neighbors is easy when the nodes have the same size, so we will consider this case first. Given a node $n_{k,p}$, we want to find all its neighbors having the same size that correspond to vertices of $\mathcal{G}_i$. Same size implies the same depth in $\mathcal{T}$, so every neighbor will have the same depth index

Function 1: `GetRGFastNeighbor()`

Data: Node $n_{k,p}$ (in $\mathcal{T}$), Node $t_{k,p}$ (in $\mathcal{T}_i$), Current node n_{k_i,p_i}

if *($\|p - p_i\|_2 - \frac{\sqrt{d}}{2}2^{k_i} \geq \alpha 2^k$ OR isLeaf($n_{k,p}$)) AND doesNotContainPath($n_{k,p}$)* **then**
  if *$n_{k,p}$ is not a ε-obstacle* **then**
    Remove all descendants of $t_{k,p}$;
  else
    Remove $t_{k,p}$ and its descendants;
else
  foreach *(m,q) index of children of (k,p)* **do**
    if *$t_{m,q}$ does not exist* **then**
      Create $t_{m,q}$ child of $t_{k,p}$;
    `GetRGFastNeighbor`$(n_{m,q}, t_{m,q}, n_{k_i,p_i})$;

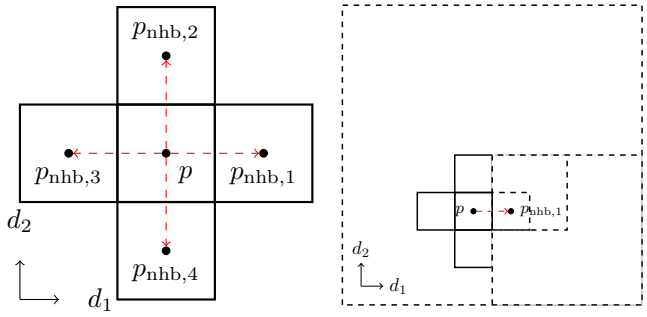

Fig. 1: (a) Generating neighbors for the construction of $\mathcal{G}_i$. Same size neighbors case: $H(n_{k,p})$ is the center square and the $p_{\text{nhb},i}$ are the generated position candidates for the neighbors; (b) Generating neighbors for the construction of $\mathcal{G}_i$. Larger neighbors case: $H(n_{k,p})$ is the smaller square around p and $p_{\text{nhb},1}$ is the first generated position candidate for the neighbors. The dashed squares represents the hypercubes corresponding to nodes visited during the search for $p_{\text{nhb},1}$ in the tree.

k. Also, the neighbor conditions and the fact that the nodes are centered on a grid, imply that only one dimension of the position vector can be changed at a time, that is, the neighbors' positions $p_{\text{nhb},i}$ can only be

$$p_{\text{nhb},i} = p + 2^k b_i, \quad 1 \leq i \leq 2d$$

with

$$b_i = \begin{cases} d_i & \text{if } i \leq d, \\ -d_i & \text{otherwise,} \end{cases}$$

where d_i is the i^{th} vector of the standard basis of $\mathbb{R}^d$. If $p_{\text{nhb},i}$ is within the bounds of the search space, $n_{k,p_{\text{nhb},i}}$ is in $\mathcal{T}_i$ and $n_{k,p_{\text{nhb},i}}$ is a leaf of $\mathcal{T}_i$, then $n_{k,p_{\text{nhb},i}}$ is a valid neighbor of $n_{k,p}$. Figure 1 shows in the center the hypercube corresponding to $n_{k,p}$ and the neighbor candidates around it. The red arrows represent the vectors $2^k b_i$.

Searching the tree $\mathcal{T}_i$ can be done on average in $O(\log |V|)$, and the number of candidates to check is $2d$.

C. Larger Neighbors

Consider now the case of finding the larger neighbors of $n_{k,p}$. The previous result can still be used, but it will generate points inside larger neighbors instead of their positions. The search through the tree works as follows. It starts with the root of the tree, representing the entire environment, as the current node. As long as the current node has children (recall that our data structure assumes that they either all exist or none of them exists), the child whose hybercube contains the searched point $p_{\text{nhb},i}$ is selected as the current node. That is, at each step, the search process selects the node at the next level of resolution whose hypercube contains $p_{\text{nhb},i}$. The search stops if either the current node is at $p_{\text{nhb},i}$ or the current node does not have children. At the end of the process, the current node is a neighbor of $n_{k,p}$ and if it is a leaf, then it is also a vertex of $\mathcal{G}_i$. A larger node could contain p and then not be a neighbor, but since $n_{k,p}$ exists, that node would have children and the search process will never stop in such a situation.

Figure 1(b) shows, in dashed lines, the last four nodes that would be explored while searching for $p_{\text{nhb},1}$. If $n_{k,p_{\text{nhb},1}}$ does not exist, the algorithm will stop at one of its ancestors, which will be a neighbor of $n_{k,p}$. Note that the search cannot stop at the largest ancestor shown, since it contains $n_{k,p}$, so all the children exist.

D. Smaller Neighbors

The last case to consider is when there are smaller neighbors of $n_{k,p}$. The search for $p_{\text{nhb},i}$ in $\mathcal{T}_i$ will return a node that is not a leaf. In this case, the exploration of

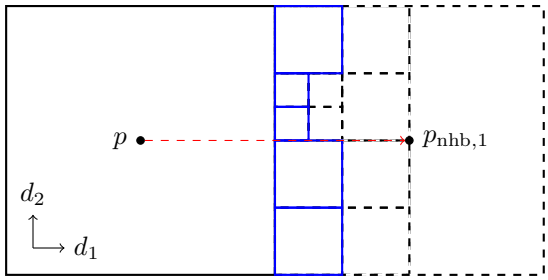

Fig. 2: Generating neighbors for the construction of $\mathcal{G}_i$. Smaller neighbors case: $H(n_{k,p})$ is the left square and $p_{\text{nhb},1}$ is the first generated position candidate for the neighbors. The larger dashed square is $H(n_{k,p_{\text{nhb},1}})$ and the blue squares correspond to the descendant of $n_{k,p_{\text{nhb},1}}$ that are neighbors of $n_{k,p}$.

children of $p_{\text{nhb},i}$ can lead to the neighbors. Note that $p_{\text{nhb},i}$ was generated by moving in the direction b_i, but since the neighbors are smaller, the move was too large, and hence neighbors of $n_{k,p}$ are leaf nodes, descendant of $p_{\text{nhb,i}}$ in the direction $-b_i$.

Figure 2 shows what happens for smaller neighbors. The larger dashed square is the hypercube corresponding to the candidate neighbor $p_{\text{nhb},1}$, but that node is not a leaf, so it is not a vertex of $\mathcal{G}_i$. Exploring its children (until leaf nodes) in the direction $-b_1 = -d_1$, will lead to all its descendants that are neighbor with $n_{k,p}$, and in $\mathcal{G}_i$, since they will be leaf nodes. The neighbors are drawn in blue in Figure 2.

E. Computing All Neighbors in $\mathcal{G}_i$

When looking for all neighbors, nodes can be treated in any order, in particular, from smallest to largest. All neighbor pairs can then be found by looking for larger neighbors for each node ordered from the smallest to the largest. Finding larger and same size neighbors is done in $O(\log|V|)$ for each of the $|V|$ nodes, so finding all neighboring pairs in $\mathcal{G}_i$ is then be done in $O(|V|\log|V|)$.

Finding all neighbors of a given node $n_{k,p}$ can be done using the pseudo-code in Function 2. For each direction b_i, we compute the candidate neighbor position $p_{\text{nhb},i}$ and search in $\mathcal{T}_i$ for the corresponding node. If the node is a leaf, it means that a larger or same size neighbor has been found; otherwise, the leaf descendants, in the direction $-b_i$, of the node found are smaller size neighbors.

Function 2: `findNeighbors()`

```
Data: Node n_{k,p}
neighbors=∅;
foreach i in [1, 2d] do
    n_{m,q}=find(T_i,p_{nhb,i});
    if isLeaf(n_{m,q}) then
        neighbors=neighbors ∪ n_{m,q};
    else
        addLeafInDir(n_{m,q}, -b_i, neighbors);
return neighbors;
```

Function 3: `addLeafInDir()`

```
Data: Node n_{k,p}, Direction b, List neighbors
foreach i in [1, 2^d] do
    if b^T e_i > 0 then
        n=child(n_{k,p},i);
        if isLeaf(n) then
            neighbors=neighbors ∪ n;
        else
            addLeafInDir(n, b, neighbors);
```

IV. Multi-Scale Path Planning Without Full Information Map - MSPP-S

Although in 2D or 3D geometric workspaces, the multi-scale map is often the result of perception algorithms, there may exist cases where we may only have access to a predicate about whether a point of the search space is an obstacle or not. A robotic arm, for example, is usually parameterized by the position of each joint; given a configuration, the spatial position of each link can be computed, and self-collision or collision with obstacles is checked in the geometric workspace. It is assumed in this section that we have such a predicate, say $isObstacle(s)$, that informs us if a point s of the search space is an obstacle.

In the proposed approach, sampling is used to estimate the obstacle probabilities of the nodes in $\mathcal{G}_i$. Since we are using an estimate instead of the exact node probability values, completeness of the algorithm is not ensured. Note, however, that if a large enough number of samples is drawn, the estimated probabilities will be close to their actual values, and loss of completeness is very unlikely. An upper bound on the probability of failure is derived in Section VI.

Similarly to the original MSPP algorithm, the proposed algorithm, MSPP-S (for MSPP with sampling), decomposes the space using a grid, which has fine resolution near the current position, and the resolution becomes increasingly coarser farther away. An empty tree data structure $\mathcal{T}_i$ is created to represent that grid. For each node $n_{k,p}$ of the partition, the predicate can be used for a given number N_{samples} of random points drawn in the search space corresponding to the node. An estimate of the probability of obstacles can then be calculated from those results and used to fill up the tree $\mathcal{T}_i$. The value of the node is approximated by

$$\hat{V}(n_{k,p}) = \frac{\text{Number of obstacles sampled}}{N_{\text{samples}}}.$$

Similarly to the data structure used for neighbor checking, it tunrs out that it is less costly to modify the data structure from the previous iteration than to recreate a new data-structure at each iteration. Moreover, in that case, some information will already exist in the data structure, and sampling only needs to be done for the newly added nodes.

The pseudo-code for the vertex selection is given in Function 4 and the edges can be computed as described in Section III-E.

Function 4: `GetRGVerticesWithSampling()`

```
Data: Node t_{k,p} (in T_i), Current node n_{k_i,p_i}
if (‖p − p_i‖_2 − (√d/2) 2^{k_i} ≥ α 2^k AND
doesNotContainPath(t_{k,p}) then
    Remove all descendants of t_{k,p};
    if n_{k,p} has not been sampled yet then
        Sample N_samples in H(n_{k,p});
        Estimate V̂(n_{k,p});
else
    foreach (m, q) index of children of (k, p) do
        if t_{m,q} does not exist then
            Create t_{m,q} child of t_{k,p};
        GetRGVerticesWithSampling(t_{m,q},n_{k_i,p_i});
```

V. Minimal Reduced Graph Construction

Constructing $\mathcal{G}_i$ can be costly and only part of the information might be used at each iteration to solve for the shortest path. In this work, it is assumed that the planning problem on $\mathcal{G}_i$ is solved using the A* algorithm although this is not restrictive. In the A* algorithm, nodes are kept in a priority queue, called $OPEN$, ordered by f-values, where $f = g+h$ with g the cost-to-go and h an admissible heuristic to the goal. While $OPEN$ has elements, the first element is removed and for each of the neighbors, if they have not been closed yet, the g-value is updated, and it is added to the

$OPEN$ priority queue. The algorithm stops when the first element of the $OPEN$ priority queue is the goal.

In the A* algorithm, knowing the neighbors of a node is only useful when that node is taken out of the $OPEN$ queue. Similarly, the obstacle probability is only needed to calculate the g-value of a node.

By delaying those calculations until the necessary information is required, improvement in execution speed is expected. The following changes allow to save computations in the new algorithm:

- the `ReducedGraph` function only computes the nodes of the reduced graph
- during the A^* algorithm, neighbors of a node are computed when the node is selected from the OPEN priority queue to be explored. If sampling is being used, sampling is only made the first time the g-value is calculated.

At the end, the algorithm will have only calculated the neighbors for the nodes in the $CLOSE$ list, and estimated the obstacle probability for nodes in $OPEN \cup CLOSE$. In the worst case, the A* algorithm will explore every vertex and every edge, so all neighbors will be calculated and all nodes will be sampled similarly to the naïve case. But, in general, the number of neighbors calculated and the number of node sampled will be largely reduced compared to the naïve case. This is confirmed by the numerical examples in the Section VII.

VI. Probabilistic bounds of MSPP-S

Since the estimate $\hat{V}(n_{k,p})$ is used instead of the actual obstacle probability $V(n_{k,p})$, a bad estimate could lead to missing solutions and losing completeness of the algorithm. In particular, if $\hat{V}(n_{k,p})$ overestimates $V(n_{k,p})$, the node $n_{k,p}$ might wrongly be evaluated as a ε-obstacle, which would prevent the algorithm from finding any path passing through $n_{k,p}$, and potentially missing the only solution, thus breaking the completeness of the algorithm. In this section, we derive an analytic worst case bound for the probability of failure of the MSPP-S algorithm.

Definition 1. *Given $\varepsilon, \gamma > 0$, a node $n_{k,p}$ is a ε, γ-obstacle if $\hat{V}(n_{k,p}) \geq 1 - 2^{-dk}\varepsilon + \gamma$.*

Let $M(n_{k,p})$ be the event that the node $n_{k,p}$ is a ε, γ-obstacle and is not a ε-obstacle. We refer the reader to [3] for the proofs of the following results.

Proposition 1. *Let $\varepsilon, \gamma > 0$ and n the number of sampled points in a node $n_{k,p}$. Then $P\big(M(n_{k,p})\big) \leq \mathrm{e}^{-2\gamma^2 n}$.*

Proposition 2. *Let a node $n_{k,p}$. If $k \geq k_{\max} = \lceil \frac{1}{d} \log_2 \frac{\varepsilon}{\gamma} \rceil$, then $n_{k,p}$ cannot be a ε, γ-obstacle and $M(n_{k,p})$ never happens.*

Proposition 3. *Let a node $n_{k,p}$. If $k \leq k_{\min} = \lfloor \frac{1}{d} \log_2 n \rfloor$, then computing $V(n_{k,p})$ is less expensive than computing $\hat{V}(n_{k,p})$. So ε-obstacles can be used and $M(n_{k,p})$ never happens.*

In the sequel we will assume that the MSPP-S algorithm uses ε, γ-obstacles when $k > k_{\min}$ and ε-obstacles otherwise. We also assume that the algorithm evaluates $\hat{V}(n_{k,p})$ at most once for each node. This value it is evaluated only when is needed and then the value is kept in memory.

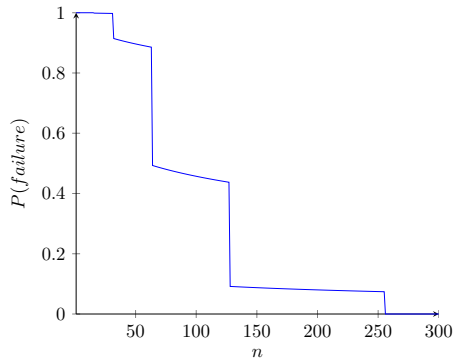

Fig. 3: Bound on the probability of failure with the parameters $\ell = 5$, $d = 1$, $\varepsilon = 90\%$, $\gamma = 0.35\%$ and $Z = 2$.

Proposition 4. *Given $\varepsilon, \gamma, n > 0$, an upper bound on the probability of failure of MSPP-S is given by*

$$P(failure) \leq 1 - \left(1 - \mathrm{e}^{-2\gamma^2 n}\right)^{nb_{\mathrm{occ}}}, \tag{3}$$

where

$$nb_{\mathrm{occ}} = \frac{2^{d(\ell - k_{\min})} - 2^{d(\ell - k_{\max}+1)}}{2^d - 1}. \tag{4}$$

Suppose now that ν independent solutions exist. Then the previous upper bound can be reduced to

$$P(failure) \leq \left(1 - \left(1 - \mathrm{e}^{-2\gamma^2 n}\right)^{nb_{\mathrm{occ}}/\nu}\right)^{\nu}. \tag{5}$$

Figure 3 shows the upper bound on the probability of failure as a function of the number of sampled points n for a given set of parameters of the algorithm. Drops correspond to an increase in $k_{\min}$, thus reducing nb_{occ}. The decrease between drops corresponds to better estimation due to more samples. As $k_{\min}$ reaches $k_{\max} - 1$, nb_{occ} goes to 0 and the probability of failure becomes 0.

These upper bounds are, of course, potentially conservative since in most cases, only part of the nodes $n_{k,p}$ with $k_{min} < k < k_{max}$ are evaluated. In addition if $M(n_{k,p})$ happens for a node that is not part of the solution, the solution will still be found by the algorithm. Finally, in typical environments, large areas of free space exists, thus multiplying the number of possible solutions and largely reducing the probability of failure. In practice, failure of the MSPP-S algorithm has not been observed.

VII. Results

A. Comparison in Random Environments

In this section, we compare the original MSPP algorithm against the proposed MSPP-FN and MSPP-S algorithms and also against A* run on a uniform grid. Obstacle maps were randomly generated and then used to solve path-planning problems via these four algorithms. The problem was solved for dimensions ranging from 2 to 5 with a tree depth of 5, that is, for search spaces ranging from $2^{2\times5} = 1024$ to $2^{5\times5} \simeq 3 \times 10^7$.

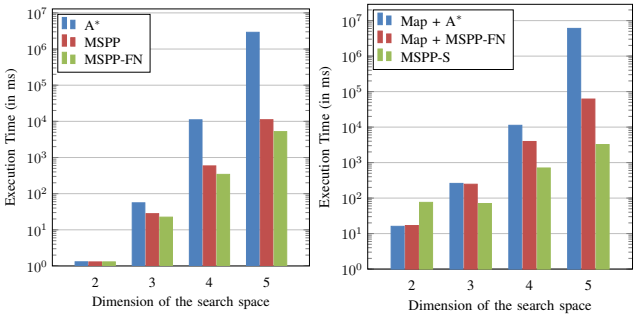

Fig. 4: (a) Comparison of the execution time of the A*, MSPP and MSPP-FN algorithms; (b) Comparison of the time to construct the map and run the A* or MSPP-FN algorithm versus the time to run the MSPP-S algorithm for which a map does not need to be computed. All results are shown in logarithmic scale.

Figure 4(a) shows the average execution time (in log scale) of the MSPP, the MSPP-FN and the A* algorithms on the randomly generated maps. In this figure, the time to create the map is not taken into account in order to compare the pure performance of the planning algorithms, that is, it is just the time to find a path on an already existing map.

For the smaller search spaces, we see very few differences between all the algorithms, as expected. As the dimension and the size of the search space grow however, the MSPP algorithm becomes much faster than the A*, by more than two orders of magnitude in dimension 5. By the same token, the MSPP-FN algorithm is even faster (by 50%) over the baseline MSPP algorithm in dimension 5.

In Figure 4(a) the cost of creating the map is taken into account. Three algorithms are compared, namely, the A* algorithm with the construction of the graph, the MSPP algorithm with the construction of the multi-scale map and the MSPP-S algorithm. Similarly to the previous case, on a small search space, there is little or no improvement. As the problem dimension increases, however, the improvement gets much better. The MSPP-S algorithm is three orders of magnitude faster than creating a map and using the A* algorithm, and more than ten times faster than constructing a multi-scale map and using the original MSPP algorithm.

B. Application to a Robot Arm

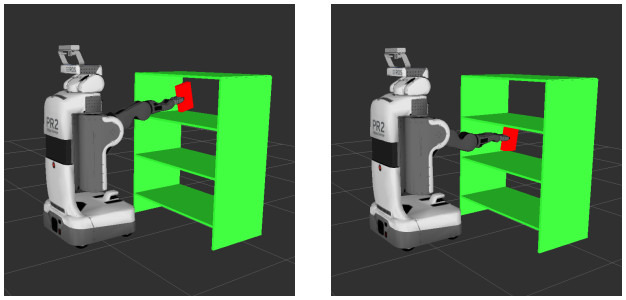

Fig. 5: Initial and final pose of the planning problem for the PR2 arm.

The planning algorithm was used to plan a trajectory for an arm of the PR2 robot. Planning was done in the configuration space using 4 joints for each arm. Figure 5 shows the initial configuration and the desired final configuration; the robot needs to move a book from the top shelf to the second shelf. The depth of the tree was set to 5, creating a search space of size $2^{4\times5} \simeq 3 \times 10^7$.

The path-planning problem was solved three times in order to compare the variants of the algorithm: First, the multi-scale map was built by exploring the entire search-space and the MSPP algorithm was used to find the solution. Building the map was the most time-consuming process. It takes on average 4 minutes and 52 seconds and solving the path-planning problem takes on average 47 seconds. Using the MSPP-FN algorithm on the same map, the problem was solved in 4 seconds on average.

VIII. Conclusions

In this paper, we have introduced several modifications and extensions to the original MSPP algorithm, first presented in [4], to increase its computational efficiency. The resulting multi-scale path-planning algorithms, called MSPP-FN and MSPP-S offer several non-trivial improvements over the previous MSPP algorithm. First, the complexity of each iteration of the algorithm is reduced by changing the manner by which the adjacency relationships in the reduced graph are computed at each iteration. Second, the range of applications of the algorithm has been broadened, by allowing the use of an obstacle predicate. Third, reordering the operations performed by the algorithm allows one to minimize computations by avoiding information that is not needed during execution. We have compared the original MSPP algorithm to the proposed MSPP-FN and MSPP-S algorithms and found runtime improvements by over 50%. Both algorithms outperform A* by more than two orders of magnitude.

References

[1] R. V. Cowlagi. *Hierarchical Motion Planning for Autonomous Aerial and Terrestrial Vehicles*. PhD thesis, Georgia Institute of Technology - School of Aerospace Engineering, 2011.

[2] F. Endres, J. Hess, J. Sturm, D. Cremers, and W. Burgard. 3-D Mapping With an RGB-D Camera. *IEEE Transactions on Robotics*, 30(1):177–187, Feb 2014.

[3] H. Florian and P. Tsiotras. Reduced complexity multi-scale path-planning on probabilistic maps. http://arxiv.org/abs/1602.04800, February 2016.

[4] F. Hauer, A. Kundu, J. M. Rehg, and P. Tsiotras. Multi-scale perception and path planning on probabilistic obstacle maps. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 4210–4215. IEEE, 2015.

[5] S. Kambhampati and L. S. Davis. Multiresolution path planning for mobile robots. *IEEE Journal of Robotics and Automation*, 2(3):135–145, 1986.

[6] Y. Lu, X. Huo, and P. Tsiotras. Beamlet-like data processing for accelerated path-planning using multiscale information of the environment. In *49th IEEE Conference on Decision and Control (CDC)*, pages 3808–3813, Atlanta, Georgia USA, December 15–17, 2010.

[7] D. K. Pai and L.-M. Reissell. Multiresolution rough terrain motion planning. *IEEE Transactions on Robotics and Automation*, 14(1):19–33, 1998.

[8] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*, Anchorage, Alaska, May 3–8, 2010.