Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)
IOS Press, 2006

Incremental Reconstruction of Generalized Voronoi Diagrams on Grids

Nidhi Kalra [1], Dave Ferguson and Anthony Stentz
The Robotics Institute, Carnegie Mellon University, Pittsburgh, USA

Abstract. We present an incremental algorithm for constructing and reconstructing Generalized Voronoi Diagrams (GVDs) on grids. Our algorithm, Dynamic Brushfire, uses techniques from the path planning community to efficiently update GVDs when the underlying environment changes or when new information concerning the environment is received. Dynamic Brushfire is an order of magnitude more efficient than current approaches. In this paper we present the algorithm, compare it to current approaches on several experimental domains involving both simulated and real data, and demonstrate its usefulness for multirobot path planning.

Keywords. Voronoi diagrams, incremental algorithms, robot navigation

1. Introduction

Efficient path planning is a fundamental requirement for autonomous mobile robots. From a single robot navigating in a partially-known environment to a team of robots coordinating their movements to achieve a common goal, autonomous systems must generate effective trajectories quickly. The efficiency of path planning algorithms can be greatly affected by the type of representation used to encode the environment. Common representations include uniform and non-uniform grids, probabilistic roadmaps, generalized Voronoi diagrams (GVDs), and exact cell decompositions. GVDs in particular are very useful for extracting environment topologies. A GVD is a roadmap that provides all possible path homotopies in an environment containing obstacle regions. The GVD also provides maximum clearance from these regions. Such a representation has practical application to many robotic domains such as multirobot planning, stealthy navigation, surveillance, and area coverage. Figure 1 presents the GVD of an outdoor environment.

GVDs can be used as complete representations of their environments [14,3] and to augment other representations such as probabilistic roadmaps [6,5]. Given a GVD, planning from a start position to a goal position consists of three steps. First, plan from the start to its closest point on the GVD (the access point). Second, plan along the GVD until the point closest to the goal is reached (the departure point). Then, plan from the departure point to the goal. Since the GVD is a graph, any graph search algorithm can be used to plan between the access and departure points, often at a fraction of the computational expense required to plan over the complete environment.

[1]Correspondence to: Nidhi Kalra; Robotics Institute; Carnegie Mellon University; 5000 Forbes Ave, Pittsburgh, PA 15232 Tel.: +1 412 268 9923; E-mail: nidhi@ri.cmu.edu

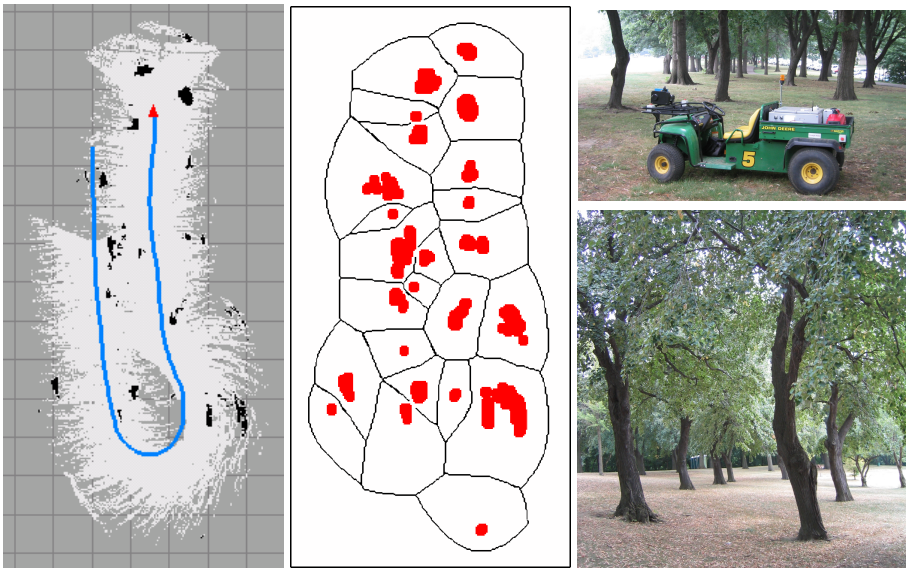

Figure 1. An outdoor environment traversed by one of our John Deere eGator robotic platforms. *(Left)* The map generated by the robot (white is traversable area, black is untraversable, gray is unknown, and the grid has 5m spacing). *(Center)* The GVD constructed from this map by Dynamic Brushfire with C-space obstacle expansion. *(Right)* The eGator and the environment in which these maps were generated.

In many domains, robots must navigate in environments for which prior information is unavailable, incomplete, or erroneous. To harness the benefits of GVDs for planning in these environments, a robot must update its map when it receives sensor information during execution, reconstruct the GVD, and generate a new plan. GVD reconstruction and replanning must occur frequently because new information is received almost continuously from sensors. However, because new information is usually localized around the robot, much of the previous GVD remains correct and only portions require repair. Unfortunately, as we discuss in Section 2, the existing algorithms for constructing GVDs have no local reconstruction mechanism and cannot take advantage of this feature; instead, they discard the existing GVD and create a new one from scratch. This frequent full reconstruction is both computationally expensive and unnecessary.

In this paper, we present the Dynamic Brushfire algorithm for incremental reconstruction of GVDs on grids. We first discuss related techniques for GVD construction. Next, we present our algorithm both intuitively and through pseudocode. We then compare this algorithm to existing algorithms on several common robot navigation scenarios on both real and simulated data. We also demonstrate the usefulness of GVDs and our algorithm in particular for coordinated multirobot path planning.

2. Related Work

The Voronoi region of an obstacle O is the set of points whose distance to O is less than or equal to their distance to every other obstacle in the environment. The GVD of an environment is the intersection of two or more Voronoi regions. Each of these intersection points is equidistant from its two (or more) closest obstacles. Several methods exist for

computing the GVD. First, the GVD can be constructed in continuous space. For example, Nageswara *et al.* [12] represent the obstacles as simple polygons and geometrically construct the GVD, and Choset *et al.* [4] simulate an agent "tracing out" the GVD as it moves through the environment. Second, the GVD can be constructed in discrete space, *e.g.* on grids. In this paper we focus on GVDs constructed on grids because of the prevalence of grid-based environment representations in mobile robot navigation. Here, the Voronoi regions and the GVD are computed over the finite set of grid cells.

Researchers have used graphics hardware to generate grid-based GVDs very quickly [7]; however, this is often infeasible for mobile robot platforms with limited on-board hardware. Alternatively, fast hardware-independent algorithms exist for constructing GVDs on low-dimensional grids [1,2,13]. These algorithms require as input a binary grid and a mapping from each occupied cell in the grid to the obstacle to which it belongs (the latter information helps determine the boundaries between different Voronoi regions). In general, these algorithms scan the grid and compute for each cell its closest obstacle and its distance to that obstacle; those cells that are equidistant from two or more obstacles are included in the GVD. These algorithms run in linear time $O(mn)$ where m and n are dimensions of the grid. Thus, the computation required is a factor of the resolution of the representation rather than the complexity of the environment.

The first of these is the well-known *Brushfire* algorithm [1]. Brushfire is analogous to Dijkstra's algorithm for path planning in that it processes cells in an *OPEN* priority queue, where decreasing priority maps to increasing distance from an obstacle. Initially, each obstacle cell in the environment is placed on the queue with a priority of 0 (the cell's distance to the nearest obstacle). Then, until the queue is empty, the highest-priority cell is removed, its neighboring cells' distances are computed, and any cell c whose distance $dist_c$ has been lowered is updated to be on the queue with priority $dist_c$. The distance $dist_c$ of each cell is approximated from the distances of its neighbors:

$$dist_c = \min_{a \in Adj(c)} [\text{distance}(c, a) + dist_a] \tag{1}$$

where $Adj(c)$ is the set of cells adjacent to c (usually 4- or 8- connected) and distance(c, a) is the distance between c and a (usually Manhattan or Euclidean distance). Brushfire only makes one pass through the grid but has the added expense of keeping a priority queue which usually requires $O(log(x))$ time for operations, where x is the number of elements in the queue.

The quasi-Euclidean distance transform algorithm developed by Rosenfeld *et al.* [13] makes two sequential passes through the grid, from top to bottom and then bottom to top. For each cell encountered, it simply performs an 8-connected search of neighboring cells to determine the cell's distance and nearest obstacle using Eq. 1. The Euclidean distance transform algorithm by Breu *et al.* [2] constructs the GVD by determining which Voronoi regions intersect each row in the grid. For each cell in a particular row, it computes the exact distance to the obstacle that is closest to it.

Although these algorithms are fast, they provide no mechanism for incremental reconstruction. We compare their performances with Dynamic Brushfire on several example robotics scenarios in Section 4.

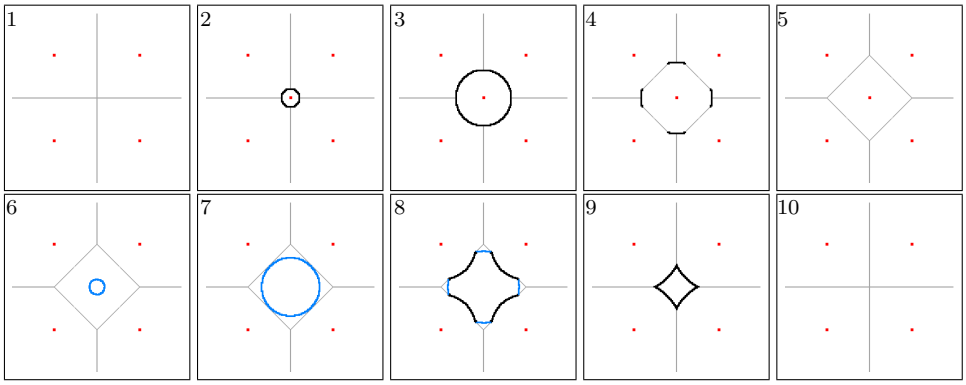

Figure 2. A sequence of distance propagations when obstacles (red/grey dots) are added and removed. *(Top)* A new obstacle cell in the center of the environment produces a wave of overconsistent states (in bold black) which terminates at the boundary of the new Voronoi region (in gray). *(Bottom)* The center cell is removed and produces a wave of underconsistent states (in bold blue/light gray) that pushes out to the corners of the old Voronoi region while an overconsistent propagation (in bold black) pulls back inwards to the boundaries of the new Voronoi regions.

3. The Dynamic Brushfire Algorithm

Just as Brushfire is analogous to Dijkstra's algorithm for planning, Dynamic Brushfire is analogous to the unfocused D* family of efficient replanning algorithms [15,10]. When new information is received concerning the environment (*e.g.* from a robot's sensors), these algorithms only propagate the new information to portions of the map that could be affected. Thus, they avoid unnecessarily re-processing the entire state space. In the grid-based GVD context, new information consists of obstacles being asynchronously added and removed from the environment, in whole or in part. When an obstacle cell is removed, the distances increase for exactly those cells that were closer to it than to any other obstacle cell. When an obstacle cell is added, the distances decrease for exactly those cells that are closer to it now than to any other obstacle cell. Dynamic Brushfire is efficient because it determines and limits processing to only cells within these two sets.

3.1. Algorithm Intuition

Dynamic Brushfire requires the same input as the other approaches discussed in Section 2: a grid and a mapping from obstacle cell to obstacle. For each cell s it maintains the obstacle $obst_s$ to which it is currently closest. It also maintains a distance $dist_s$ to its closest obstacle from the previous GVD construction and a one-step lookahead distance $dist_s^{new}$ to the closest obstacle given the changes that have occurred since that construction. A cell is consistent if $dist_s = dist_s^{new}$, overconsistent if $dist_s > dist_s^{new}$, or underconsistent if $dist_s < dist_s^{new}$.

Like A*, D*, and Brushfire, Dynamic Brushfire keeps a priority queue *OPEN* of the inconsistent cells to propagate changes. A cell s's priority is always $\min(dist_s, dist_s^{new})$ and cells are popped with increasing priority values. When a cell is popped from the *OPEN* queue, its new distance is propagated to its adjacent cells, and any newly-inconsistent cells are put on the *OPEN* queue. Thus, the propagation emanates from the source of the change and terminates when the change does not affect any more cells.

When a cell is set to obstacle, it reduces the new minimum distance of adjacent cells which propagate this reduction to their adjacent cells, etc. This creates an overconsistent sweep emanating from the original obstacle cell (Figure 2, panels 1 to 3). An overconsistent sweep terminates when cells are encountered that are equally close or closer to another obstacle and thus cannot be made overconsistent. These cells lie on the boundary between Voronoi regions and will be part of the new GVD (Figure 2, panels 4 and 5).

When an obstacle cell is removed, all cells that previously used it to compute their distances are invalid and must recompute their distances. This propagation occurs in two sweeps. First, an underconsistent propagation sweeps out from the original cell and resets the affected cells (Figure 2, panels 6 and 7). This sweep terminates when cells are encountered that are closer to another obstacle and cannot be made underconsistent. Thus, at most those cells in the removed obstacle cell's Voronoi region are invalided. Then, an overconsistent propagation sweeps inwards and uses the valid cells beyond this region to recompute new values for the invalid cells (Figure 2, panels 8 and 9).

3.2. Algorithm Pseudocode

Initialize ()

```
1 OPEN =TIES = ∅;
2 foreach cell s
3     dist_s =dist_s^new =∞
4     parent_s =tie_s =∅
5     obst_s =∅
```

SetObstacle (o, O)

```
1 dist_o^new = 0
2 obst_o = O
3 parent_o =o
4 valid_O =TRUE
5 insert(OPEN , o, dist_o^new)
```

RemoveObstacle (o, O)

```
1 dist_o^new =∞
2 obst_o =∅
3 if O - o =∅
4     valid_O =FALSE;
5 insert(OPEN , o, dist_o)
```

Figure 3. Pseudocode for initializing the grid and adding and removing obstacle cells.

Figures 3, 4, and 5 present pseudocode for the Dynamic Brushfire algorithm. In addition to $dist_s$, $dist_s^{new}$, and $obst_s$, each cell s tracks the cell $parent_s$ from which its distance is computed and a cell tie_s that is adjacent to s and equidistant to a different obstacle. Additionally an obstacle is invalid ($valid_O$ = FALSE) when it is entirely removed from the environment. Dynamic Brushfire first initializes all distances to infinite and pointers to unknown. Then, obstacles are added and removed using SetObstacle and RemoveObstacle. When a cell o is added to obstacle O (SetObstacle(o, O)), its distance becomes zero, its parent cell is itself, and it is placed on the queue as an overconsistent cell. When a cell o is removed from obstacle O (RemoveObstacle(o, O)), its properties are reinitialized and the cell is placed on the queue as an underconsistent cell.

The GVD is reconstructed by calling the function RebuildGVD which removes cells from the *OPEN* queue for processing. When an overconsistent cell is removed (lines 2-4), its lookahead distance will be correct, so its current distance is updated. When an underconsistent cell is removed (lines 7-10), its old distance is reset so that it can subsequently be processed as an overconsistent cell.

The GVD is marked in two steps. First, when a cell s becomes consistent (line 3 in RebuildGVD), we check if it should be considered for inclusion in the GVD. In the function ConsiderForGVD, if a cell s has an adjacent cell n in a different Voronoi region, s is placed on the *TIES* list of possible GVD cells and n is recorded for later use. Thus, if any two adjacent cells lie in different Voronoi regions, at least one of them will be placed on the *TIES* list. Once the *OPEN* list is empty, the *TIES* list is examined in function ConstructGVD to determine which cells belong on the GVD using the same criterion.

```
RebuildGVD()
 1 while s = remove(OPEN )
 2     if dist_s^new < dist_s
 3         dist_s = dist_s^new;
 4         ProcessLower(s)
 5         ConsiderForGVD(s)
 6     else
 7         dist_s = ∞
 8         if dist_s^new ≠ dist_s
 9             insert(OPEN , s, dist_s^new)
10         ProcessRaise(s)
11 ConstructGVD()

ConsiderForGVD(s)
 1 foreach n ∈ Adj(s)
 2     if obst_s ≠ obst_n
 3         tie_s = n
 4         insert(TIES , s)

ConstructGVD(s)
 1 while s = remove(TIES )
 2     foreach n ∈ Adj(s)
 3         if obst_n ≠ obst_s
 4             n and s are on the GVD
 5             tie_s = n ; tie_n = s
 6         else
 7             n and s are not on the GVD
 8             tie_s = ∅ ; tie_n = ∅
```

Figure 4. Pseudocode for rebuilding the GVD. Function RebuildGVD removes cells from the *OPEN* queue to be processed and functions ConsiderForGVD and ConstructGVD mark the GVD.

```
ProcessLower(s)
 1 foreach n ∈ Adj(s)
 2     if tie_n = s
 3         insert (TIES , n)
 4     if dist_n^new > dist_s^new
 5         d'= distance(n, s) + dist_s^new)
 6         if d'< dist_n^new
 7             dist_n^new = d'
 8             parent_n = s
 9             obst_n = obst_s
10             insert(OPEN , n, dist_s^new)

ProcessRaise(s)
 1 foreach n ∈ Adj(s)
 2     if tie_n = s
 3         insert(TIES , n)
 4     if parent_n = s
 5         dist_n^old = dist_n^new ; dist_n^new = ∞
 6         obst_n^old = obst_n
 7         foreach a ∈ Adj(n) s.t. obst_a is valid
 8             d'=distance(n, a) + dist_a^new
 9             if d'< dist_n^new
10                 dist_n^new = d'; parent_n = a
11                 obst_n = obst_a
12         if dist_n^new ≠ dist_n^old or obst_n ≠ obst_n^old
13             insert(OPEN , n, min(dist_n^new, dist_n))
```

Figure 5. Pseudocode for propagating overconsistencies (ProcessLower) and underconsistencies (ProcessRaise).

The functions ProcessLower and ProcessRaise both update cells' distances and place inconsistent cells on the queue. ProcessLower propagates overconsistencies: a cell examines its neighbors to see if it can lower their distances. Any changed neighboring cell has its lookahead distance, parent, and closest obstacle updated and is placed on the queue. ProcessRaise propagates underconsistencies: a cell examines its neighbors to see if any used its old distance to compute their own (line 4). For each such cell, the lookahead distance is reset and then recomputed using current values (lines 7-11). Any changed cell is placed back on the *OPEN* list. Additionally, both ProcessLower and ProcessRaise also find neighboring cells that were marked on the GVD because their Voronoi region was different from the current cell's. Such cells are reinserted into the *TIES* list to be reexamined (lines 2-3). The termination and correctness of Dynamic Brushfire follow from the proof of D* Lite [11] and are shown in [8].

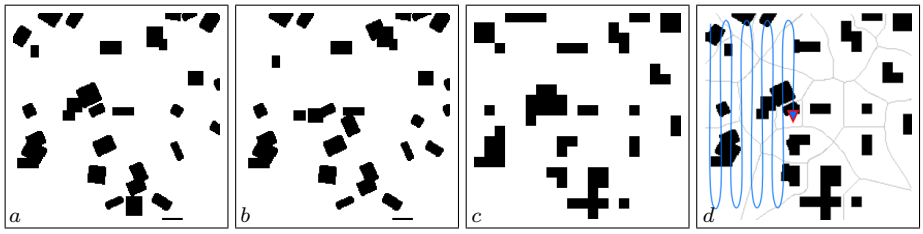

Figure 6. The maps used to generate results. From left to right, the complete correct map, the map with random errors, the map at lower resolution, and an illustration of an Unmanned Aerial Vehicle (UAV) updating the GVD to the correct map from the low resolution map.

4. Experiments and Results

In this section we present and discuss statistical comparisons between Dynamic Brushfire and competing algorithms on several robotic scenarios. We also present results from applying Dynamic Brushfire to real robot data and to a multirobot planning problem.

4.1. Comparison to Other Algorithms

We compared Dynamic Brushfire to the Brushfire, Euclidean Distance Transform, and quasi-Euclidean Distance Transform algorithms discussed in Section 2 on four scenarios. The first scenario is common to many domains and involves constructing the GVD once given a static environment. We use 200×200 cell environments that are 20% occupied by randomly placed obstacles ranging in size from 5×5 to 20×20 cells (see Figure 6 *(a)*). The remaining three scenarios are unique to and occur often in robotics: they require traversing the environment and repairing the GVD as new information is gathered from sensors. For this, we simulate an Unmanned Aerial Vehicle (UAV) sweeping this environment with a 10 cell omni-directional sensor and require that the UAV repair the GVD after every 10 cells of traverse. In the first of these three scenarios, the UAV has no prior map. In the second, it must repair the GVD given an erroneous prior map in which each obstacle has a 30% chance of having incorrect dimensions, of being placed incorrectly, or of being absent. An additional 10% of obstacles are randomly added (see Figure 6 *(b)*). Finally, the third involves repairing a GVD given a 20×20 cell low-resolution prior map (see Figure 6 *(c)*). Figure 6 *(d)* illustrates the UAV's traverse as it incorporates new information into a prior low-resolution map. We ran each algorithm on each scenario on 100 different maps.

The results from these experiments are shown in Table 1 and in Figure 7. The performance difference between Dynamic Brushfire and the other algorithms depends primarily on how much of the old computation can be preserved and how often repair must occur. This is true of most incremental repair algorithms (*e.g.* D* and D* Lite). Thus, in initial construction of the GVD (where prior computation does not exist), the extra processing that enables repair causes Dynamic Brushfire to be competitive to but slightly slower than the other algorithms.

In the other three scenarios, Dynamic Brushfire outperforms its competitors by an order of magnitude. This improvement is most pronounced when updating from the low resolution and erroneous maps (Dynamic Brushfire is 20 and 17 times faster, respectively, than its closest competitor) because information about the environment is gained at every step and is localized around the robot. When constructing the GVD without a prior map,

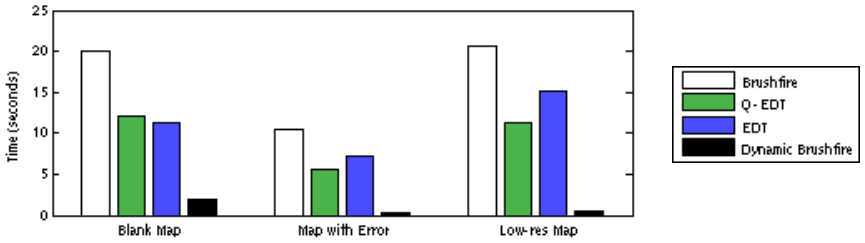

Figure 7. A graph of the results from Table 1 on the three incremental update scenarios. From left to right in each group: Brushfire, EDT, Q-EDT, and Dynamic Brushfire.

Table 1. A comparison of the performances of the four approaches on four GVD construction and repair scenarios: initial construction of a complete map (Initial), incremental construction without a prior map (Blank), incremental construction with an erroneous prior map (Error), and incremental construction with a low resolution prior map (Lowres). Each scenario was tested 100 times for each algorithm; the left column indicates the mean total time taken in seconds, and the right column indicates the standard deviation of this measure.

Algorithm	Initial		Blank		Error		Lowres	
	μ	σ	μ	σ	μ	σ	μ	σ
Brushfire	0.158	0.006	20.075	1.091	10.467	2.382	20.666	1.393
Q-EDT	0.091	0.003	12.161	0.061	5.662	1.256	11.356	0.065
EDT	0.104	0.003	11.264	0.078	7.227	1.669	15.212	1.276
DynamicBrushfire	0.162	0.005	1.887	0.174	0.328	0.051	0.551	0.068

large portions of the map have yet to be observed and the GVD is sparse. Thus, changes must be propagated over larger areas than in the other cases. As a result, the improvement is significant (Dynamic Brushfire is 6 times faster) but less than in the other scenarios.

4.2. GVD Construction on Real Data

We have also tested Dynamic Brushfire on real environmental data obtained from traverses by a John Deere e-Gator robotic platform equipped with a SICK LMS laser in an outdoor wooded environment (Figure 1, top and bottom right). In one experiment, our robot explored a 75 $\times$ 40 meter area (Figure 1, left). We generated a GVD of the final data (Figure 1, center). We also repaired the GVD after each sensor scan from the laser. The roughly 2,400 cell updates over 90 repair episodes in a 375×200 cell environment took Dynamic Brushfire 2.7 seconds. For comparison, the same experiment took Brushfire 31.1 seconds.

In a second experiment, we gathered sensor data from a traverse by the e-Gator platform in a similar but larger environment and then returned three weeks later for a second traverse over the same area. The first traverse provided a prior map for the second traverse. Here, repair on a prior map of size 680 $\times$480 cells with roughly 2,200 cell updates over 156 repair episodes took Dynamic Brushfire 11.4 seconds. For comparison, the same experiment took Brushfire 250.6 seconds.

4.3. Application to Multirobot Path Planning

This work was originally motivated by the need for efficient path planning for multirobot teams. Specifically, we are interested in solving the constrained exploration problem in which a team of agents must navigate through an environment while maintaining com-

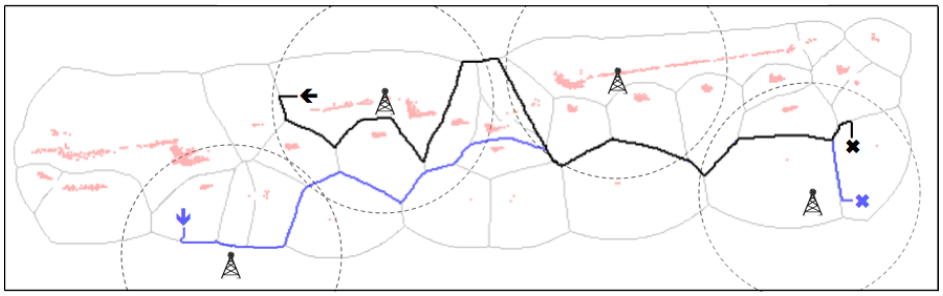

Figure 8. The path of two robots successfully completing a constrained exploration task from the left to the right of the environment. The paths are in bold blue/gray and bold black, obstacles are in light gray, and the GVD generated on this environment is in thin light gray. The coverage areas of the four communication towers are marked by dashed circles.

munication constraints. An instance of this problem is presented in Figure 8. Here, two robots must navigate through the environment to reach their goal locations, from the left to the right. Each robot must also always remain in contact with one of the communication towers in the environment at all times, either directly or indirectly via its teammate. Because the area of coverage of some neighboring communication towers do not overlap, the robots must coordinate to hop over the gaps. In such domains where teammates' actions are highly interdependent, centrally planning for parts of the team can be beneficial [9]. However, algorithms for computing centralized plans typically have complexity exponential in the number of robots. While normally such planning might be intractable, it can be made feasible by reducing the search space (*e.g.* by planning only on the GVD).

We compared planning on the GVD to planning over the entire space. In this instance, A* took only 0.36 seconds to determine a path when the search space was limited to the GVD, while it took 94.9 seconds when searching over the entire grid. We then repeated the same task but this time gave the robots an erroneous prior map of the environment. They constructed the GVD of the current map, planned a path, executed a portion of that plan while improving the map with sensor information, and then repaired the GVD and replanned again. Replanning after every five steps resulted in 74 planning and GVD construction episodes; the total planning time was 12.2 seconds and the GVD repair time using Dynamic Brushfire was 4.5 seconds. For comparison, GVD repair using Brushfire took 132.3 seconds.

5. Conclusions

In this paper we have presented Dynamic Brushfire, a new algorithm for efficient, incremental reconstruction of GVDs on grids. Dynamic Brushfire operates analogously to the D* family of algorithms for path replanning: it restricts the propagation of changes in the environment to only those areas that could be affected. We have compared our algorithm to several leading approaches for constructing GVDs and found our algorithm to be significantly more efficient, particularly in typical robotics applications. We are currently using this algorithm for multirobot path planning in unknown and uncertain environments.

6. Acknowledgments

This work was sponsored by the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance" (contract number DAAD19-01-2-0012). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements of the U.S. Government. Dave Ferguson is supported in part by an NSF Graduate Research Fellowship.

References

[1] Jerome Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. Technical Report STAN-CS-89-1257, Computer Science Department, Stanford University, May 1989.

[2] Heinz Breu, Joseph Gil, David Kirkpatrick, and Michael Werman. Linear time euclidean distance transform algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:529–533, May 1995.

[3] Howie Choset and Joel Burdick. Sensor based planning, part I: The generalized voronoi graph. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1995.

[4] Howie Choset, Sean Walker, Kunnayut Eiamsa-Ard, and Joel Burdick. Sensor-based exploration: Incremental construction of the hierarchical generalized voronoi graph. *The International Journal of Robotics Research*, 19(2):126 – 148, February 2000.

[5] Mark Foskey, Maxim Garber, Ming Lin, and Dinesh Manocha. A voronoi-based hybrid motion planner. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2001.

[6] Leonidas Guibas, Christopher Holleman, and Lydia Kavraki. A probabilistic roadmap planner for flexible objects with a workspace medial-axis-based sampling approach. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1999.

[7] Kenneth Hoff, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 1999.

[8] Nidhi Kalra, Dave Ferguson, and Anthony Stentz. The dynamic brushfire algorithm. Technical Report CMU-RI-TR-05-37, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, September 2005.

[9] Nidhi Kalra, Dave Ferguson, and Anthony Stentz. Hoplites: A market-based framework for complex tight coordination in multi-robot teams. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2005.

[10] Sven Koenig and Maxim Likhachev. D* lite. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2002.

[11] Maxim Likhachev and Sven Koenig. Lifelong Planning A* and Dynamic A* Lite: The proofs. Technical report, College of Computing, Georgia Institute of Technology, 2001.

[12] Nageswara Rao, Neal Stoltzfus, and S. Sitharama Iyengar. A "retraction" method for learned navigation in unknown terrains for a circular robot. *IEEE Transactions on Robotics and Automation*, 7(5), October 1991.

[13] Azriel Rosenfeld and John Pfaltz. Sequential operations in digital picture processing. *Journal of the Association for Computing Machinery*, 13(4), 1966.

[14] Peter Forbes Rowat. *Representing spatial experience and solving spatial problems in a simulated robot environment*. PhD thesis, University of British Columbia, 1979.

[15] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1994.