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

Solving Shortest Path Problems with Curvature Constraints Using Beamlets

Oktay Arslan, Panagiotis Tsiotras and Xiaoming Huo

***Abstract*— A new graph representation of a 2D environment is proposed in order to solve path planning problems with curvature constraints. We construct a graph such that the vertices correspond to feasible beamlets and the edges between beamlets capture not only distance information but directionality as well. The A* algorithm incorporated with new heuristic and cost function is implemented such that the curvature of the computed path can be constrained a priori. The proposed Beamlet* algorithm allows us to find paths with more strict feasibility guarantees, compared to other competing approaches, such as Basic Theta* or A* with post-smooth processing.**

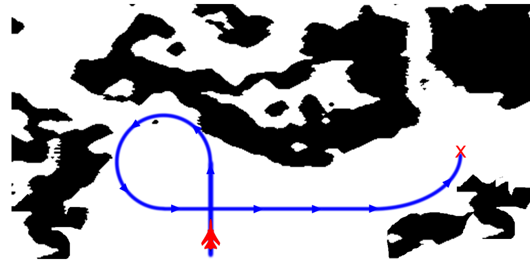


Fig. 1: Solution of the problem may be a self-intersecting curve due to constraints

I. Introduction

Planning a path for an autonomous vehicle in a 2D or 3D environment has been studied for many years [1], [2] and many algorithms have been developed to solve this problem [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14] to name just a few. From a system-theoretic view point, the path planning problem is to find a control input that will drive the state of the vehicle from a given initial state to a given goal state, while minimizing a cost function and satisfying the state and vehicle dynamic constraints. This is a complex problem since it requires searching a solution in an infinite dimensional space. One of the most common approach to overcome this difficulty is to decompose the original problem into two subproblems which address the geometric and dynamic parts separately. However, the feasibility of the computed path in the geometric level does not always imply that the computed path can be followed by the vehicle. Since the geometric level planner does not have any prior information about the dynamic envelope of the vehicle, this approach may lead to either a suboptimal or a dynamically infeasible path. Generation of dynamically infeasible paths can be avoided by coupling the geometric and dynamic level planners via passing information about the allowable dynamic envelope of the vehicle during the geometric search. One simple approach to capture the vehicle's dynamic envelope is to bound the curvature of the allowable paths, and pass this information to the geometric level planner [15].

Path planning algorithms that solve the problem at the geometric level can find obstacle-free paths (no kinematic and dynamic constraints) very easily. These algorithms usually search on a graph which is generated by discretizing the continuous environment into cells that are either blocked (black) or unblocked (white). A common decomposition is a grid-like partitioning to square cells. The corners of these cells and the links among them correspond to vertices and edges of the search graph, respectively. As a result of this abstraction, paths which are formed by grid edges are not smooth since the possible headings at each vertex are artificially constrained; furthermore, the graph formed by the grid is insufficient to express all possible paths which are embedded in the environment. For instance, consider a path planning problem for the vehicle shown in Figure 1, where the destination is shown with the cross and the vehicle can turn only to the left due to a malfunction of its actuators. It is obvious that the trajectory of the vehicle should be a self-intersecting path, as shown in Figure 1. But this path cannot be computed by any search algorithm (Dijkstra, A* or any variants) at this level of abstraction (i.e., one that searches on the graph formed by the topological grid of the environment [16], [17]). This occurs because these algorithms do not allow for self-intersecting paths. One solution to avoid such problems is to increase the graph dimension to also account for extra states (such as velocity) at each graph vertex, however this increases the dimensionality of the problem. Here we follow an alternative approach. We still perform the path-planning in the physical space (where the obstacles are more naturally described), but we also encode local directionality of the path in an efficient manner. It turns out that this is possible using beamlets [18]. Beamlets are small line segments connecting points at the boundary of cells in a dyadic partitioning of the environment, organized in a hierarchical structure. They provide an efficient encoding of all smooth curves in the plane with an error that is no larger than the underlying resolution, in terms of Hausdorff distance [18, Lemma 1]. In other words, they offer sparse

Oktay Arslan is a graduate student in the D. Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA, email: oktay@gatech.edu

Professor Panagiotis Tsiotras is with the D. Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA, email: tsiotras@gatech.edu

Professor Xiaoming Huo is with the H. Milton Stewart School of Industrial and System Engineering, Georgia Institute of Technology, Atlanta, GA, 30332-0250, USA, email: huo@gatech.edu

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

approximate representations of smooth curves in the plane; in a certain sense, they are optimally sparse. As shown in [18], the library of all beamlets is of order $O(n^2 \log_2 n)$, whereas all possible segments connecting the $O(n^2)$ vertices in the grid (i.e., the *beams*) are of order $O(n^4)$. The latter order of complexity is prohibitive for the development of an efficient algorithm. The reduced cardinality of beamlets, on the other hand, allows us to develop algorithms that have complexity only logarithmically larger than the given data. It follows that exhaustive searches through the collection of beamlets can be implemented much faster than exhaustive searches through the collection of beams.

In this paper we use the previous remarkable property of beamlets, to efficiently connect points between empty spaces in the plane[1], and−most importantly−to include directionality information along the ensuing path. This will allow us to easily incorporate curvature constraints along the path. Such curvature constraints are meant to provide a "dynamic signature" for the resulting path so that can be followed by the actual vehicle.

II. Related Work

Several different algorithms have been proposed to find smooth trajectories in a continuous environment, notably among them are the A* with Post Smoothed Paths [19] and the Basic Theta* [20]. Both A* with Post Smoothed Paths (A* PS) and the Basic Theta* algorithms perform some extra operations (line-of-sight check) in order to remove redundant vertices from the optimal path. The A* PS algorithm first computes the shortest path using the classical A* search algorithm, and then it performs a post-processing operation to smooth the computed path. During the post-processing operation, a vertex is removed from the computed path if its successor vertex is in the line-of-sight of its predecessor vertex. Its successor vertex is then assigned as the parent of its predecessor vertex. On the other hand, the Basic Theta* algorithm performs a line-of-sight check during the update procedure of the each vertex. It assigns the parent of the current vertex as the parent of the neighbor vertex if the neighbor vertex is in the line-of-sight of the parent of the current vertex. Although both A* PS and Basic Theta* can find smooth paths, they perform the smoothing operation and the removal of the redundant vertices only around a neighborhood of the optimal shortest path found by A*. They cannot apply curvature constraints during the search and expansion of the vertices. The proposed approach, on the other hand, uses a new representation of the information of the environment in a way such that *a priori* curvature constraints can be imposed directly during each step of the search. Therefore, it can find smoother paths with a guaranteed curvature bound.

III. Approach

The path planning problem we are interested in is to find a *curvature bounded path* from a start point (p_{start}) to a goal point (p_{goal}) while minimizing a predefined cost function. It is assumed that the spatial information about the environment is given as an $n \times n$ square image, where $n = 2^{s_{\max}}$ and $s_{\max}$ is a positive integer. There are several key elements which play an important role in the formation of the beamlet graph. Their definition is given below.

Beamlets is a collection of line segments which are defined between certain points in the environment, selected from the boundaries of dyadic squares [18], [21]. Unlike the points on the grid, beamlets may not necessarily have uniform properties and they occupy different scales and orientation on a given image as shown in Figure 2(a). A beamlet is *feasible* if all cells it passes through are unblocked. A *dyadic square* (d-square) is a set of points in the environment forming a square region. Each dyadic square is characterized by a 3-tuple (scale s and position (x,y)). Formally, it is defined as $q(s;x,y) = \{(i,j) : 2^s(x-1) \leq i \leq 2^s x, 2^s(y-1) \leq j \leq 2^s y\}$. Furthermore, $q(s;x,y)$ can be rewritten as the union of four sub-d-squares of scale $s-1$; i.e., we have $q(s;x,y) = q(s-1;2x-1,2y-1) \cup q(s-1;2x-1,2y) \cup q(s-1;2x,2y-1) \cup q(s-1;2x,2y)$. This property of d-squares leads to a recursive partitioning scheme which is called *dyadic partitioning*.

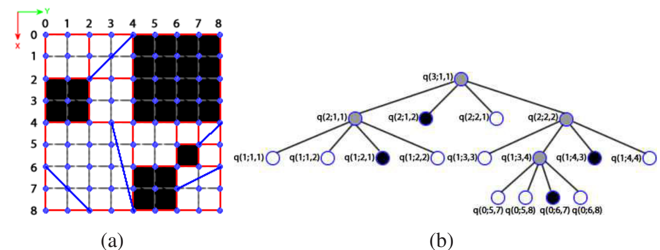


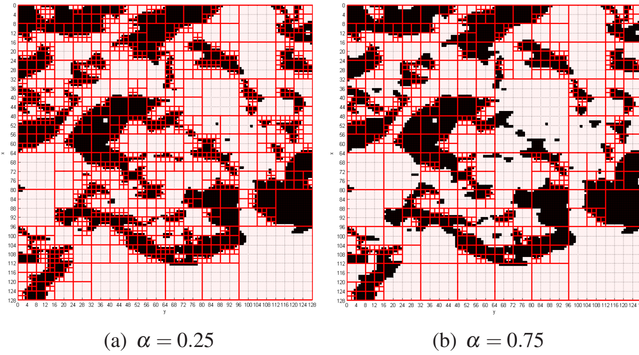
Fig. 2: Sample of feasible beamlets on a 8×8 square image and the corresponding quadtree representation.

Due to the recursive nature of the d-squares, the set of all d-squares of a given dyadic partition can be stored as a quadtree [22]. Figure 2(b) illustrates the corresponding quadtree of the dyadic partition given in Figure 2(a).

A. Multiresolution Representation of the Environment

A quadtree decomposition is used to obtain a multiresolution representation of the environment, as shown in Figure 3. The algorithm basically counts the number of obstacles in each dyadic square in an efficient manner. If the counted number of obstacles is in the interval $[1+\alpha(2^{s-1}-1), 2^s - 1 - \alpha(2^{s-1}-1)]$, where s and α are the scale of the dyadic square and tuning parameter, respectively, it divides the current dyadic square further. This algorithm continues partitioning until there are no dyadic square which can be divided any further.

[1]While two vertices in the nearest-neighbor graph corresponding to vertices at opposite sides of the environment can only be connected by a path with $O(n)$ edges, no two vertices in the beamlet graph are ever more than $4\log_2(n)+1$ edges apart.



(a) $\alpha = 0.25$ (b) $\alpha = 0.75$

Fig. 3: Quadtree decomposition of the environment for two different values of obstacle distribution.

Typically, the nodes of the quadtree are labeled as white (free of obstacles), black (full of obstacles), or gray (mixture of both free cells and obstacles), and the leaves of the quadtree are allowed to be only either white or black. In our implementation, this condition is relaxed and the quadtree is allowed to have gray leaves. The tendency of the quadtree to have gray leaves is controlled by the parameter $\alpha \in [0,1]$. The greater the value of α, the more the tendency to have gray leaves in the quadtree. The algorithm creates a quadtree with no gray leaves if $\alpha = 0$; the quadtree has only the root node if $\alpha = 1$.

B. Construction of the Beamlet Graph

The beamlet graph $\mathscr{BG} \triangleq (\mathscr{V}, \mathscr{E})$ is defined such that each element in the set of vertices $\mathscr{V}$ corresponds to a feasible beamlet. Two beamlets are geometrically connected if they share a common point. The beamlet neighborhood of a given beamlet i includes all beamlets which are geometrically connected to the end points of beamlet i, as shown in Figure 4. The edge set $\mathscr{E} \subset \mathscr{V} \times \mathscr{V}$ consists of all pairs (i, j) where $i, j \in \mathscr{V}$, such that the beamlets i and j are geometrically connected. Each edge $e_{ij} \in \mathscr{E}$ in $\mathscr{BG}$ includes two types of information: the length of the beamlet j and the angle between the two beamlets as shown in Figure 5.

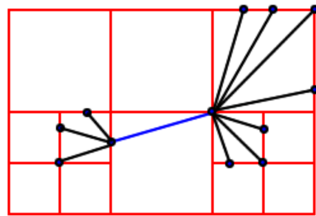


Fig. 4: Beamlet connectivity.

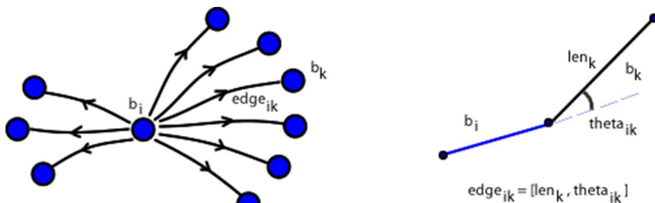


Fig. 5: Edge between beamlets

After a multiresolution representation of the environment is given, we find the set of all feasible beamlets that are geometrically attached to one of the end points of a given beamlet b. First, the algorithm finds the set of dyadic squares which include one of the end points of the beamlet b. The black dyadic squares are ignored since they are full of obstacles and do not include any feasible beamlets. For each boundary point of a white dyadic square, a beamlet emanating from one of the related end points of the beamlet b is created. There is no need to check its feasibility, since there are no obstacles in a white dyadic square. For gray dyadic squares, in addition to the creation of the corresponding beamlets, a feasibility check is required since the dyadic square is mixture of free and blocked cells. This algorithm helps to build the beamlet graph incrementally as needed during the search.

C. Description of Search Algorithm

Several functions are defined in order to apply the search algorithm on the beamlet graph. An edge cost function $D : \mathscr{E} \to \mathbb{R}_+ \times [-\pi, \pi]$ maps each edge in the beamlet graph to a pair of distance and an angle (ℓ, θ). A path-cost function $G : \mathscr{V} \to \mathbb{R}_+ \times \mathbb{R}_+ \times [0, \pi]$ maps each vertex to a cost vector whose components are ($\sum \ell$, $\sum |\theta|$, $|\theta|_{\max}$). $\sum \ell$ is the sum of the length of all beamlets corresponding to the path from the start vertex to the current vertex, $\sum |\theta|$ is the sum of the absolute value of the angles between the beamlets of the path, and $|\theta|_{\max}$ is the maximum of the absolute values of those angles. A path-cost function $add(c, d)$ updates the path-cost vector c with the given edge cost d. It adds the edge cost $d = (\ell, \theta)$ to the current value of the path-cost information $c = (\sum \ell, \sum |\theta|, |\theta|_{\max})$ and updates $|\theta|_{\max}$ if $|\theta|$ is greater than its current value. This function is given in Algorithm 2. Since the path-cost information is a vector, the function $g(s)$ is defined in line 11 of Algorithm 2 in order to compare the "goodness" of different path-cost vectors. The function g maps each path-cost vector to a single non-negative cost value. The cost value is a linear combination of the distance ℓ and angle θ , scaled appropriately.

The precedence relation $\prec (c', c)$ between two path-cost vectors c and c' is defined as follows: c' precedes c if and only if g_{score} of c' is smaller than g_{score} of c. Finally, a heuristic $h : \mathscr{V} \to \mathbb{R}_+$ is defined in line 7 in Algorithm 2 to guide the search over the beamlet graph. The heuristic is admissible since it considers the cost information (ℓ, θ) of a beamlet formed by the second point of the current beamlet s_{p_2} and the goal point p_{goal}. Simple geometry ensures that these cost value contributions will never overestimate the actual ones.

The A* search algorithm with Fibonacci heap [23], [24] as priority queue is used to search for the optimal path on the beamlet graph. For any given start and goal points (p_{start},p_{goal}), the algorithm first calls $Initialize(p_{\text{start}}, p_{\text{goal}})$ in line 4 in Algorithm 1 to compute a multiresolution representation of the environment and initialize internally used data structures. Then, the $ComputeBeamletsEmanating(\mathfrak{BG}, p)$ function is called at lines 5 and 6 to create a set of start and goal vertices in the beamlet graph. This function first

Algorithm 1: Beamlet*

```
1  Beamlet*(p_start, p_goal, |θ|_max)
2      frontier ← ∅
3      explored ← ∅
4      BG ← Initialize(p_start, p_goal)
5      s_start ← ComputeBeamletsEmanating(BG, p_start)
6      s_goal ← ComputeBeamletsEmanating(BG, p_goal)
7      foreach s' ∈ s_start do
8          parent(s') ← null
9          c'_Σℓ ← ||s'_p2 − s'_p1||
10         c'_Σ|θ| ← c'_|θ|max ← 0
11         G(s') ← c'
12         frontier.insert(s', g(s') + h(s'))
13     while frontier ≠ ∅ do
14         s ← frontier.pop()
15         if s ∈ s_goal then
16             return "pathfound"
17         explored.insert(s)
18         foreach s' ∈ neighbor(s) do
19             if s' ∉ explored then
20                 d ← D(edge(s,s'))
21                 if |d_θ| ≤ |θ|_max then
22                     if s' ∉ frontier then
23                         g(s') ← ∞
24                     UpdateState(s,s')
25 UpdateState(s,s')
26     d ← D(edge(s,s'))
27     c ← G(s)
28     c' ← G(s')
29     c_new ← add(c,d)
30     if c_new ≺ c' then
31         parent(s') ← s
32         G(s') ← c_new
33         if s' ∈ frontier then
34             frontier.remove(s')
35         frontier.insert(s', g(s') + h(s'))
```

finds the set of dyadic squares which include the point p by traversing the quadtree and it then computes a set of feasible beamlets by attaching the point p to the points located at the boundary of the dyadic squares. The path-cost vector and parent of the each vertex in the set of start vertices s_{start} are initialized in line 7 to 11. The first and second points of a given beamlet s' are represented by s'_{p_1} and s'_{p_2}, respectively in line 9. Those vertices are inserted into the $frontier$ heap. The rest of the search procedure is the same as in the A* algorithm, except that during the vertex expansion step, the algorithm checks whether the transition violates the given curvature constraint before updating a neighbor vertex at line 23. If the angle value θ of the edge cost d is greater than $|\theta|_{\max}$, then that neighbor vertex is ignored. As a result, although two vertices are geometrically connected, the transition between them is not allowed due to the violation of the curvature constraint.

Algorithm 2: Heuristic, Cost and Other Functions

```
1  add(c,d)
2      c_Σℓ ← c_Σℓ + d_ℓ
3      c_Σ|θ| ← c_Σ|θ| + |d_θ|
4      if c_|θ|max < |d_θ| then
5          c_|θ|max ← |d_θ|
6      return c
7  h(s)
8      s' ← Beamlet(s_p2, p_goal)
9      d ← D(edge(s,s'))
10     return β * d_ℓ * n_ℓ + (1 − β) * |d_θ| * n_θ
11 g(s)
12     c ← G(s)
13     g_score ← β * c_Σℓ * n_ℓ + (1 − β) * c_Σ|θ| * n_θ
14     return g_score
15 ≺(c',c)
16     g'_score ← β * c'_Σℓ * n_ℓ + (1 − β) * c'_Σ|θ| * n_θ
17     g_score ← β * c_Σℓ * n_ℓ + (1 − β) * c_Σ|θ| * n_θ
18     return g'_score < g_score
```

IV. Comparison with Existing Methods

We compared the proposed Beamlet* algorithm with the A*, the A* with Post Smoothed Paths (A* PS) and the Basic Theta* algorithms on several graphs formed by an eight-neighbor square grid. All search algorithms were implemented in C++ on a 2.5 GHz Core 2 with 3 GB of RAM laptop. All algorithms solved the same problems on several artificially created 2D cluttered environments which are characterized by two different grid sizes and 4 different percentages of occurrence of obstacles. For a given percentage of obstacles, some cells in the environment were randomly blocked in order to create a cluttered environment. For each environment, 100 path planning problems were created by choosing the start and goal points randomly. The computed paths were compared with respect to the following criteria, averaged over 100 path planning problems: the length of the computed path ℓ, the absolute value of maximum heading angle $|\theta|_{\max}$, the execution time of the search algorithm t_e, the number of expanded vertices during the search n_{exp}, and the number of heading changes along the path n_{hc}. We used quadtree decompositions where $\alpha = 0$ for all cases, that is, there are no gray leaves in the final quadtree. Finally, the values of β, n_ℓ, and n_θ were set to 1.0, $\sqrt{2}n$, and π, where n is the dimension of the square which represents environment, respectively. We tried to find the shortest possible path which satisfies the given heading angle constraint.

First, Beamlet* was used to solve a path planning problem whose only possible solution is a self-intersecting curve owing to a constraint on the heading angle along the path. This is shown in Figure 6, where only left turns (positive heading angles) are allowed along the path. For this problem,

the start and goal points are the top-left and bottom-right green points, respectively, as shown in Figure 6. None of the A*, A* PS and Basic Theta* algorithms can find the solution for this problem. This is due to the fact that these algorithms will start expanding all vertices at the top-left area of the map from the start point, but will never reach the goal point after all vertices are expanded. Since each vertex which corresponds to a point in the graph is never re-expanded once it has been inserted in the *explored* list, the path computed by A*, A* PS and Basic Theta* algorithms will never pass through the same point twice.

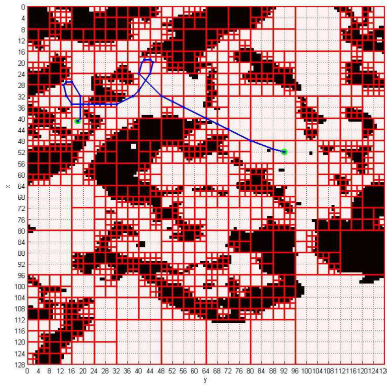


Fig. 6: Beamlet* searches over a larger space of curves and can find self-intersecting paths.

The comparison of these algorithms with respect to the path length and the maximum heading angle is summarized in Table I. Other metrics related to the computed paths are illustrated in Figures 7-9. Figure 7 shows the number of heading changes along the corresponding paths, Figure 8 shows the number of expanded vertices, and Figure 9 shows the execution times of the search algorithms.

As seen in Table I, Beamlet* can find smoother paths than Basic Theta* and the other algorithms for all cases tested, as expected. This is owing to the fact that Beamlet* allows us to explicitly constraint the upper and lower bounds of the heading changes along the path. Therefore, although a vertex in the beamlet graph has too many geometrically connected neighbor vertices, only the ones that have an edge which does not violate the heading angle bounds are updated during the search.

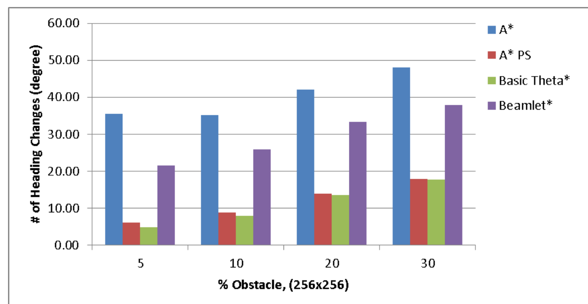


Fig. 7: Heading Changes (grid size: 256x256)

The number of heading changes on the paths computed by Beamlet* is usually larger than those computed by other algorithms, as shown in Figure 7. The reason is that Beamlet* computes multiple soft maneuvers instead of a single sharp maneuver due to the constraint on the maximum absolute value of the heading angle. Also, it is obvious that the number of heading changes on a path depends on how cluttered the environment is, and how the obstacles are distributed inside the environment. Therefore, in our numerical comparison, the constraint on the heading angle was relaxed on more cluttered environments in order to increase the chance of finding a path. We constrained $|\theta|_{\max}$ to 15°, 20°, 25° and 30° when executing the Beamlet* algorithm for path planning problems in the environments of 5%, 10%, 20%, and 30% obstacles, respectively.

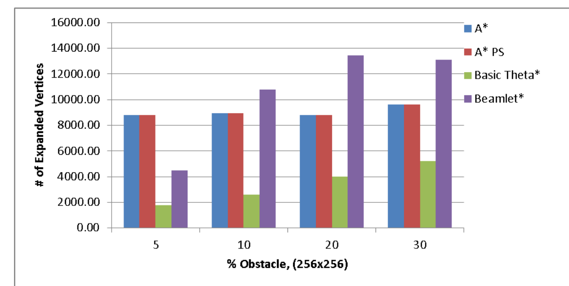


Fig. 8: Vertex Expansions (grid size: 256×256)

The Beamlet* algorithm usually expands more vertices than the Basic Theta* algorithm, as shown in Figure 8. There are two main reasons for this overexpansion of vertices: First, the beamlet graph is larger than a graph formed by just the grid points, in terms of both the number of vertices and the number of edges. Second, although, the start and goal points may be spatially very close to each other in a path planning problem, and perhaps there exists a short path between the two, the solution path will be naturally longer if one imposes additional constraints on the allowable heading angle changes along the path. Computing such a longer path over a larger graph will inevitably require a larger number of vertex expansions during the search.

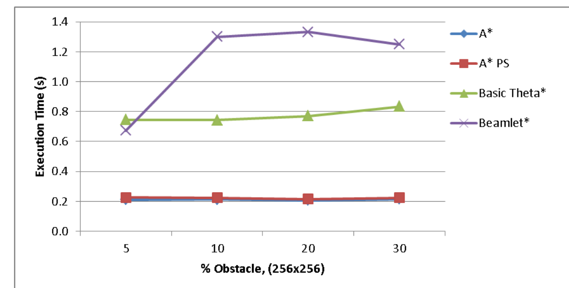


Fig. 9: Execution Time (grid size: 256x256)

Despite the large number of vertex expansions in the Beamlet* algorithm, the execution time of Beamlet* is reasonable, and sometimes even better than the Basic Theta* if the environment is less cluttered, as seen in Figure 9. This is because Beamlet* and Basic Theta* use different vertex update procedures during the search. Basic Theta* requires line-of-sight checking for each neighbor vertex of the current

	Maps	A*		A* PS		Basic Theta*		Beamlet*	
		ℓ	$\|\theta\|_{\max}$	ℓ	$\|\theta\|_{\max}$	ℓ	$\|\theta\|_{\max}$	ℓ	$\|\theta\|_{\max}$
128x128	%5 Obstacle	114.521	47.701	109.895	25.919	108.643	13.518	114.191	**9.503**
	%10 Obstacle	113.443	52.202	108.906	36.491	107.618	25.354	112.886	**13.492**
	%20 Obstacle	112.905	55.352	108.298	39.914	107.380	33.994	121.778	**14.788**
	%30 Obstacle	107.509	90.253	102.601	73.511	101.373	70.507	105.544	**26.771**
256x256	%5 Obstacle	231.993	52.652	224.841	37.849	220.310	22.445	225.348	**12.191**
	%10 Obstacle	231.341	56.702	224.023	44.627	219.430	33.305	236.584	**13.182**
	%20 Obstacle	234.551	82.352	225.681	63.228	222.767	54.608	232.998	**21.739**
	%30 Obstacle	241.613	92.253	231.788	71.666	228.910	61.383	230.576	**27.240**

TABLE I: Beamlet* computes smoother paths than Theta* as seen in $|\theta|_{\max}$ column.

vertex during vertex expansion step. This extra operation contributes a significant portion to the total execution time. On the other hand, Beamlet* only requires to check the feasibility of a beamlet if it is included by a gray dyadic square.

V. Conclusions

We have proposed a new algorithm (called Beamlet*) to compute shortest paths on a graph such that curvature constraints along the path can be easily incorporated. Such local curvature constraints serve as dynamic information surrogates to obtain feasible trajectories, and thus can bridge the gap between the geometric path-planning and feasible trajectory generation levels in the motion planning hierarchy. Compared to other similar approaches, the most important advantage of Beamlet* is that it guarantees an a priori upper bound of the heading angle changes along the resulting path. This tends to result in smoother paths. On the other hand, the result of the Beamlet* algorithm is dependent on the multiresolution representation of the environment, and the algorithm may not find a solution given a bad dyadic partition. In this work, only a simple heuristic based on the Euclidean distance was used, and a one-directional A* search algorithm was implemented. Significant improvements in performance can be achieved by incorporating different types of heuristics (e.g., landmark [25] and search algorithms (e.g., bidirectional [8], [26], [27]. These refinements of Beamlet* can increase its numerical efficiency. Finally, the proposed approach can be easily extended to path planning problems in the 3D world. In this case, the multiresolution representation of the environment will consist of dyadic cubes (instead of squares) of different scales, and the computed feasible beamlets will be stored in an octree (instead of an quadtree) data structure.

References

[1] J. C. Latombe, *Robot Motion Planning*. Springer Verlag, 1990.

[2] S. M. LaValle, *Planning Algorithms*. Cambridge Univ Pr, 2006.

[3] H. Noborio, T. Naniwa, and S. Arimoto, "A quadtree-based path-planning algorithm for a mobile robot," *Journal of Robotic Systems*, vol. 7, no. 4, pp. 555–574, 1990.

[4] C. Warren, "Fast path planning using modified a* method," in *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*. IEEE, 1993, pp. 662–667.

[5] A. Yahja, A. Stentz, S. Singh, and B. Brumitt, "Framed-quadtree path planning for mobile robots operating in sparse environments," in *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, vol. 1. IEEE, 1998, pp. 650–655.

[6] M. Jansen and M. Buro, "Hpa* enhancements," in *Third Artificial Intelligence and Interactive Digital Entertainment Conference*. Stanford, CA: The AAAI Press, June 2007, pp. 84–87.

[7] R. Holte, M. Perez, R. Zimmer, and A. MacDonald, "Hierarchical a*: Searching abstraction hierarchies efficiently," in *Proceedings of the National Conference on Artificial Intelligence*. Citeseer, 1996, pp. 530–535.

[8] A. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 156–165.

[9] A. Goldberg, H. Kaplan, and R. Werneck, "Reach for a*: Efficient point-to-point shortest path algorithms," in *Proceedings of the eighth Workshop on Algorithm Engineering and Experiments and the third Workshop on Analytic Algorithmics and Combinatorics*, vol. 123. Society for Industrial Mathematics, 2006, p. 129.

[10] S. Koenig and M. Likhachev, "D^* lite," in *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2002, pp. 476–483.

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

[12] S. M. LaValle and J. Kuffner, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, p. 378, 2001.

[13] E. Frazzoli, M. A. Dahleh, and E. Feron, "Real-time motion planning for agile autonomous vehicles," in *American Control Conference, 2001. Proceedings of the 2001*, vol. 1. IEEE, 2002, pp. 43–49.

[14] D. Ferguson and A. 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*, 2005.

[15] R. V. Cowlagi and P. Tsiotras, "Shortest distance problems in graphs using history-dependent transition costs with application to kinodynamic path planning," in *American Control Conference, 2009. ACC'09.* IEEE, 2009, pp. 414–419.

[16] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[17] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[18] D. L. Donoho and X. Huo, "Beamlets and multiscale image processing," *Multiscale and Multiresolution Methods*, vol. 20, pp. 149–196, 2001.

[19] C. Thorpe and L. Matthies, "Path relaxation: Path planning for a mobile robot," in *OCEANS 1984*. IEEE, 1984, pp. 576–581.

[20] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: Any-Angle Path Planning on Grids," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 22, no. 2. Citeseer, 2007, p. 1177.

[21] D. L. Donoho and X. Huo, "Beamlet pyramids: A new form of multiresolution analysis, suited for extracting lines, curves, and objects from very noisy image data," *Proceedings of SPIE*, vol. 4119, pp. 434–444, 2000.

[22] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

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

[24] T. H. Cormen, *Introduction to Algorithms*. The MIT press, 2001.

[25] A. Felner, N. Sturtevant, and J. Schaeffer, "Abstraction-based heuristics with true distance computations," in *Proceedings of SARA*, vol. 9, 2009.

[26] D. de Champeaux and L. Sint, "An improved bidirectional heuristic search algorithm," *J. ACM*, vol. 24, no. 2, pp. 177–191, 1977.

[27] I. Pohl, "Bi-directional search," *Machine Intelligence*, vol. 6, pp. 127–140, 1971.