Motion Planning : Incremental Densification

Aditya Vamsikrishna
(avmandal)

Amit Bansal
(ab1)

Professor
Siddhartha Srinivasa

1 Introduction

Most difficult problems can be broken down to a set simpler problems whose solutions can be used to solve or focus the solution procedure of the original problem. Algorithms that employ incremental densification are motivated by the key idea of incrementally focusing computation while solving for the motion planning problem. An analogy of this can be drawn from some established Computer Vision algorithms where a coarser image is scanned for features which is computationally simpler before focusing search at these features for more efficient feature tracking. This document draws a parallel to such computer vision algorithms in a graph-search motion planning setting. These incremental algorithms tend to be *anytime* algorithms. An interesting question to think about in this setting is if it reasonable to ask for an anytime algorithm which finds the same optimal solution as an optimal algorithm (given enough time?).

2 Algorithms and Discussion

Algorithm-1 and Fig.(1) describe a basic algorithm that employs incremental densification.

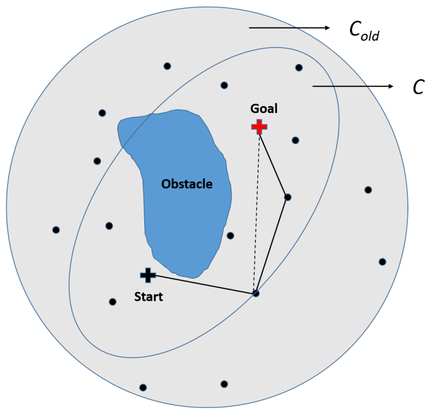

Figure 1: Incremental Densification: $\mathcal{C}_{\text{old}}$ updated to $\mathcal{C}$

One of the state of the art algorithms that uses incremental densification is BIT* algorithm [1] (Batch Informed Trees; available on OMPL) which shall be described shortly. Some of the natural questions that arise out of these algorithms are:

1. What type of graph to build in $\mathcal{C}$ when sampling points.
2. How can we efficiently reuse computation from previous iterations.

Algorithm 1 `incremental_densification` ($\mathcal{C}$)

```
1: Sample points sparsely in $\mathcal{C}$
2: **if** shortest path $\xi^*$ exists **then**
3:     $\mathcal{C} \leftarrow$ Updated $\mathcal{C}$, ellipsoid determined by $\xi^*$
4: **if** termination_condition($\xi^*$) **then**
5:     **return** $\xi^*$
6: incremental_densification ($\mathcal{C}$)
```

Random Geometric Graphs (RGG) are generally created in the configuration space due to certain computational advantages they provide. They have been extensively used in other areas like social network analysis and owe their analysis to percolation theory.

3 Random Geometric Graphs

In graph theory, a random geometric graph (RGG) is the mathematically simplest spatial network, namely an undirected graph constructed by randomly placing N nodes in some metric space (according to a specified probability distribution) and connecting two nodes by a link if and only if their distance is in a given range, e.g. smaller than a certain neighborhood radius, r.

In 1 dimension, one can study RGGs on a line of unit length (open boundary condition) or on a circle of unit circumference. In 2 dimensions, an RGG can be constructed by choosing a flat unit square [0, 1] or a torus of unit circumferences [0, 1] as the embedding space. The simplest choice for the node distribution is to sprinkle them uniformly and independently in the embedding space. We construct a random graph G(n,r) as follows. We pick vertices X1, . . . , Xn [0, 1] i.i.d. (independent, identically distributed) uniformly at random and we join Xi,Xj (i = j) by an edge if

$$||Xi - Xj|| <= r$$

Often the RGG is defined in arbitrary dimension d, with the points X1, . . . , Xn i.i.d. according to some (general) probability measure on Rd, and where the distance between points is measured by an arbitrary norm on Rd (often the lp-norm for some p)

Expected degree of a typical vertex is approximately:
np for G(n,p)
nr$^2 forG(n,r)$

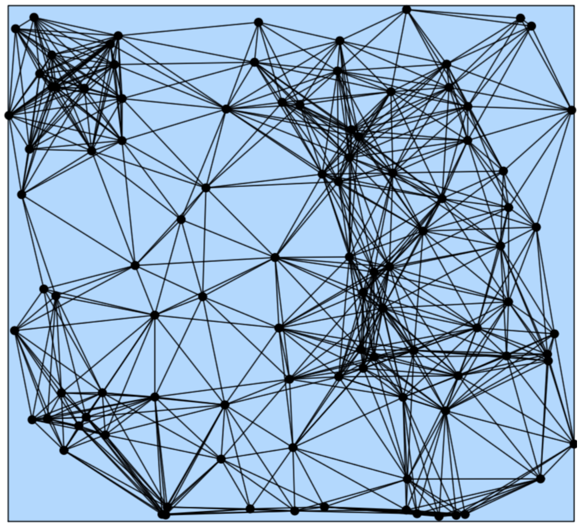

Figure 2: Random Geometric Graph

Algorithm 2 `lazy_shortest_path`

```
1: $E_{\text{eval}} \leftarrow \emptyset$
2: $w_{\text{lazy}} \leftarrow w_{\text{est}}(e) \forall e \in E$
3: while True do
4:     $p_{\text{candidate}} \leftarrow$ ShortestPath$(G, u, w_{\text{lazy}})$
5:     if $p_{\text{candidate}} \subseteq E_{\text{eval}}$ then
6:         return $p_{\text{candidate}}$
7:     $E_{\text{selected}} \leftarrow$ edge_selector$(G, p_{\text{candidate}})$
8:     for $e \in E_{\text{selected}}\ E_{\text{eval}}$ do
9:         $w_{\text{lazy}} \leftarrow w(e)$                                    ▷ Expensive
10:        $E_{\text{eval}} \leftarrow E_{\text{eval}} \cup e$
```

Algorithm 3 `edge_selector`$(G, p_{\text{candidate}})$

```
1: function SelectExpand$(G, p_{\text{candidate}})$
2:     $e_{\text{first}} \leftarrow$ first unevaluated edge $e \in p_{\text{candidate}}$
3:     $v_{\text{frontier}} G$.source$(e_{\text{first}})$
4:     $E_{\text{selected}} \leftarrow G$.out$_{\text{e}}$dges$(v_{\text{frontier}})$
5:     return $E_{\text{selected}}$
6: function SelectForward$(G, p_{\text{candidate}})$
7:     return first unevaluated edge $e \in p_{\text{candidate}}$
8: function SelectReverse$(G, p_{\text{candidate}})$
9:     return last unevaluated edge $e \in p_{\text{candidate}}$
10: function SelectAlternate$(G, p_{\text{candidate}})$
11:     if LazySP iteration is odd then
12:         return first unevaluated edge $e \in p_{\text{candidate}}$
13:     else
14:         return last unevaluated edge $e \in p_{\text{candidate}}$
15: function SelectBisection$(G, p_{\text{candidate}})$
16:     return unevaluated edge $e \in p_{\text{candidate}}$ furthest from nearest evaluated edge
```

4 LazySP Algorithm

LazySP[2] encapsulates a class of shortest path algorithms which evaluate edges in the graph for collisions *lazily*. This is motivated by many robotic applications in which the majority of the planning time and computational cost is attributed to collision checks or edge evaluations. The algorithm attempts to minimize the number of edge evaluations while searching for the shortest path in the graph.

The algorithm maintains a list of evaluated edges as E_{eval}. The algorithm also has access to w_{est} which is an optimistic estimate of the true edge weight an is inexpesive to compute. The algorithm also defines a lazy weight function w_{lazy} which returns the true weight of an evaluated edge and otherwise uses the inexpensive estimator w_{est}. As described in Algorithm 2, LazySP determines the shortest path with the current lazy weight function it has access to. It then iteratively selects edges along this path in a certain way defined by the `edge_selector` function and evaluates the edges which is an expensive operation. The algorithm repeats until the current shortest path has all the edges belonging to E_{eval}.

The LazySP algorithm proposes multiple edge selectors for efficiency of search, some of which give rise to algorithms equivalent to other A* variants such as [3] and [4]. Some of the simple edge selectors are described in Algorithm 3

References

[1] Jonathan D. Gammell, Siddhartha S. Srinivasa, Timothy D. Barfoot (2015). *Batch Informed Trees (BIT*): Sampling-based Optimal Planning via the Heuristically Guided Search of Implicit Random Geometric Graphs*. IEEE International Conference on Robotics and Automation

[3] Christopher M. Dellin, Siddhartha S. Srinivasa (2016). *A Unifying Formalism for Shortest Path Problems with Expensive Edge Evaluations via Lazy Best-First Search over Paths with Edge Selectors.* In International Conference on Automated Planning and Scheduling.

[4] Peter Hart, Nils J Nilsson, Bertram Raphael (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics

[5] Benjamin Cohen, Mike Phillips and Maxim Likhachev (2014), *Planning Single-arm Manipulations with n-Arm Robots*, Proceedings of the Robotics: Science and Systems Conference (RSS).