Bounded Diverse Paths and their application in Path Planning

Ana Huamán Quispe and Mike Stilman

Abstract We present a formal definition of Bounded Diverse Paths and how this approach can be applied to Path Planning.

Our approach rests on principles of Digital Imagery as well as Optimal Control. We show its application to a practical problem, such as how to find diverse paths.

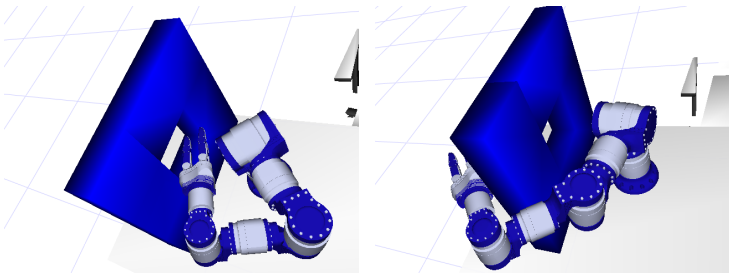
1 Introduction

Motion planning problems usually refer to the canonical problem of computing collision-free paths from a start to a goal position. Great efforts have been focused in developing techniques to produce *optimal paths*, according to some cost metric. While this is useful in some scenarios, it is also true that there are situations in which it would be more useful to produce more than one possible path as a solution. We illustrate this better as an example:

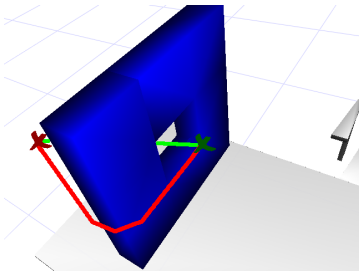
Assume that you have a robotic arm such as in Fig.1(a) and you want to devise a path to reach the goal position in Fig.1(b). Say that you want to solve this problem by using *search-based techniques* for high-dimensional spaces, such as [**?**], which make use of a *workspace heuristic of the end effector* to guide the configuration space search. If we use a standard optimal planner to obtain the optimal path in terms of length, we would get the green path depicted in Fig.1(c). However, from Fig.1(b), it is easily seen that the path depicted in red would be more appropiate to describe what the real end effector path would look like.

The red path would probably not be found by a standard planner since its longer than the optimal path. Hence, in this problem it makes sense for a planner to produce both paths and let the user decide which one is fit to the problem requirements. In order to produce both paths, we must establish that they are different in some way, or belong to *different classes*.

Center for Robotics and Intelligent Machines, Georgia Institute of Technology,Atlanta GA,
e-mail: ahuaman3@gatech.edu, mstilman@cc.gatech.edu



(a) Start Position (b) Target position



(c) Possible Paths

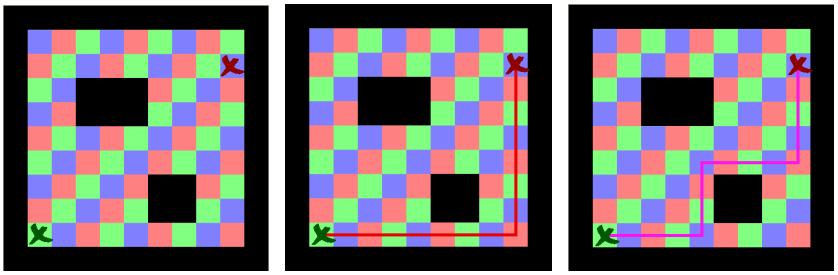
Fig. 1 Application example: An optimal planner would find the green path, however the red one would be more useful

Homotopy classes are a straightforward concept to classify paths. Two paths are homotopic if one path can be continuously deformed into another without passing through an obstacle. A homotopy class is a collection of homotopy paths. In two-dimensional environments, each obstacle generates at least a new homotopy class (ask Mike this) as it can be seen in the example at Fig.2.

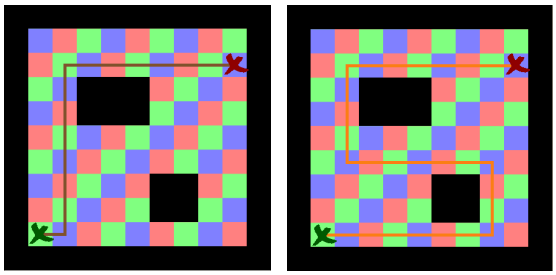
In 3D environments, the specification of paths belonging to different homotopy classes is harder to determine, since only obstacles that contain *holes* or obstacles stretching to infinity in two directions determine different homotopy classes ([**?**]). In Fig.1 the obstacle had one hole, so there were two homotopy classes, namely the set of paths that passed through the hole and the set that passed outside the obstacle. Now, think what would happen if the obstacle would have the shape shown in Fig.3. The scenery is the same as in Fig. 1, with the exception that in Fig. 3 the obstacle is not close-shaped anymore. Three paths are shown in Fig. 3(b), all of them belonging to the same homotopic class.

From the example before, we see that describing paths based on their homotopy class is, for some cases, too strict and hence it does not help much in the generation of paths that are as *diverse* as possible.

On this paper, we will propose a simple and intuitive method to automatically generate set of paths such that these are as *diverse* as possible, making it possible to consider more than one alternative.

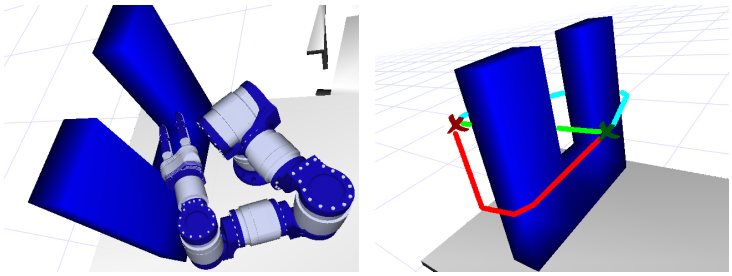


(a) Initial problem (b) Path in Homotopy class A (c) Path in Homotopy class B



(d) Path in Homotopy class C (e) Path in Homotopy class D

Fig. 2 Homotopy classes in 2D environments



(a) Problem proposed (b) Possible paths, all belonging to the same homotopic class

Fig. 3 Application example: To classify the paths by their homotopy class is not very useful here

2 Related Work

Efforts to produce different paths have been mostly related to homotopy classification. The problem of homotopy in 2D environments has been throroughly studied and there are diverse solutions present in robotics literature. The approaches are diverse, ranging from geometric methods, such as []. Later on, Schmitzberger worked on the problem by using Probabilistic Roadmaps ([**?**]), obtaining a planner that was able to capture paths from different homotopy classes, however these paths were not optimal, due to the intrinsic randomized nature of the PRM. 5 Bhattacharya et

al presented an interesting method to find optimal paths with homotopic path constraints [**?**], based on a weird rule that I still don't get but has something to do with my Electrical undergrad courses, for sure. Igarashi and Stilman ([**?**]) on the other hand, presented a simple and powerful algorithm that also generate optimal paths on different homotopy classes by using a variant of greedy search with a heuristic that build overlapping manifolds corresponding to the different homotopic classes.

Path generation with homotopic constraints in 3D environments has been recently studied in [**?**] by using more weird electrical laws that are extensive to any 3D scene (i.e. a 2D dynamic environment in which the time t is the third dimension).

As we saw in the introduction, Homotopy alone is not enough to get the variety we are looking for.

Here we can cite work of Likhachev et al [**?**], and stuff about Distance Transform such as the work of Pedro Fzelnzab as well as the application of Distance Transforms to Path Planning, although its goal was different than ours

Mention coverage stuff by Zelinsky [**?**]

Problem pretty much solved by Stilman [**?**] and Likhachev [**?**]

Why it is hard and recent stuff from Knepper [**?**] and roots from Simeon [**?**] and Jaillet [**?**]

3 Definitions

Definition 1 (Search Space). We define a general *search space* as an undirected connected graph $\mathscr{G}$, with a set $\mathscr{V}$ of vertices and a set $\mathscr{E}$ of edges. In this paper, we further consider:

- $\mathscr{V}$: A finite path-connected set of discretized *free* space locations (free voxels). $\mathscr{V} \in \mathbb{N}^3$
- $\mathscr{E}$: A finite set of connections between two neighboring vertices $v_i, v_j \in \mathscr{V}$.

From the definition above, it is easy to see that occupied locations (i.e. obstacles) are not represented in $\mathscr{G}$.

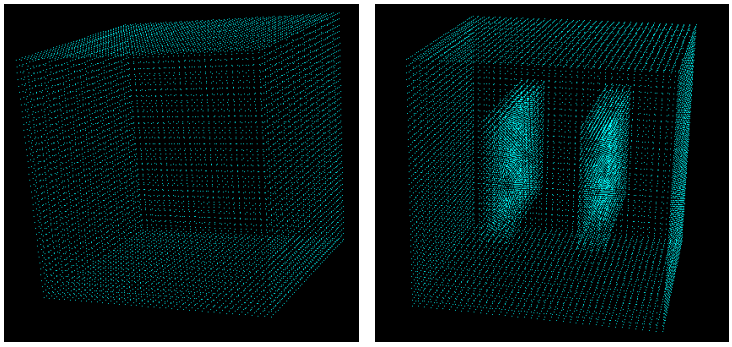
Examples of Search Spaces are shown in Fig.4. Fig.4(a) show a uniformly gridded box, where $\mathscr{V}$ is comprised of the free space voxels inside the colored grid. Fig.4(b) shows a more complex space in which obstacles inside the box were added.

Definition 2 (Path). Given the set $\mathscr{V}$ of vertices from $\mathscr{G}$ and two vertices v_a and v_b, we define a path P as a sequence of vertices such that:

$$P = \{p_1, p_2, ..., p_n\}$$

such that:

- $p_1 = v_a$, $p_n = v_b$
- $(p_i, p_{i+1}) \in \mathscr{E}, \forall p_i \in P$, $i \in [1, n-1]$



(a) Path Connected Set (b) Connected Set

Fig. 4 Examples of Discrete Connected Sets

additionally, we define the cardinality of P:

$$| P | = n$$

Definition 3 (Metric Space). We define the *metric space* $\mathscr{M}$ based on a given search space $\mathscr{G}$ as:

$$\mathscr{M} = (\mathscr{V}, d_{min}) \tag{1}$$

where

$$d_{min} : \mathscr{V} \times \mathscr{V} \to \mathbb{R}$$

d_{min} is the length of the shortest path connecting any vertices v_i, $v_j \in \mathscr{V}$. d_{min} holds the following properties:

1. $d_{min}(x,y) \geq 0$
2. $d_{min}(x,y) = 0 \iff x = y$
3. $d_{min}(x,y) = d_{min}(y,x)$
4. $d_{min}(x,z) \leq d_{min}(x,y) + d_{min}(y,z)$

Definition 4 (Vertex-Set Distance). Consider a search space $\mathscr{G}$ and its corresponding metric space $\mathscr{M}$. Define a set of vertices S as:

$$S = \{s_1, s_2, ..., s_{n_s}\}, s_i \in \mathscr{V}$$

The distance from any vertex $q \in \mathscr{V}$ with respect to S as the *set distance* D_{VS} :

$$D_{VS}(q,S) = \inf_{i=[1,...,n_s]} \{d_{min}(q,s_i)\}$$

Definition 5 (Added Distance). Given a path $P \in \mathscr{V}$ and a set $S \in \mathscr{V}$. We define the *added distance* (D_{added}) of P with respect to S as:

$$D_{added}(P,S) = \sum_{i=0}^{|P|} D_{VS}(p_i, S)$$

Based on Definition 5, a special case occurs when the set S is the union of a set of paths $\mathscr{P}$:

$$\mathscr{P} = \bigcup_{i=1}^{k} P_i = P_1 \cup P_2 \cup ... \cup P_k$$

Then, the Added Distance from a path P_{k+1} with respect to this set of paths $\mathscr{P}$ is:

$$D_{added}(P_{k+1}, \mathscr{P}) = \sum_{i=0}^{|P_{k+1}|} D_{VS}(p_{k+1\,i}, \mathscr{P}) \tag{2}$$

Now we have all the needed definitions to formally enunciate our problem:

Definition 6 (Diverse Paths Problem).

For a search space $\mathscr{G}$ we define a start and target locations, represented by the vertices v_{start} and $v_{target} \in \mathscr{V}$. We want to find a set of paths $\mathscr{P}$ that hold the following conditions:

For a search space $\mathscr{G}$ we define start and target vertices v_{start} and $v_{target} \in \mathscr{V}$. We want to find a *sequence* of k paths:

$$\mathscr{P} = (P_1, P_2, P_3..., P_k)$$

such that they hold the following properties:

- $\forall P_i \in \mathscr{P}, p_{i1} = v_{start}$ and $p_{in} = v_{target}$
- P_1 must be the shortest path from v_{start} to v_{target}
- The paths belonging to $\mathscr{P}$ must be bounded in length with respect to P_1. Formally:
$$\forall P_i \in \mathscr{P}, |P_i| \leq \alpha \, |P_1| \, , \alpha \geq 1, \forall i \in < 1, k] \tag{3}$$
This condition imposes a constraint on the maximum length of candidate paths
- Each path P_i is as far as possible from its predecessors in $\mathscr{P}$. The metric to express this distance is the D_{added}:
$$P_i = \arg\max_{P_i} \{D_{added}(P_i, \{P_1, ...P_{i-1}\})\} \tag{4}$$

In the next section, we present an algorithm to systematically build $\mathscr{P}$.

4 Algorithm

Let us formulate the problem as an *optimization problem*, and then we will intuitively derive the algorithm presented.

Assume you have two functions $f(.)$ and $g(.)$ defined on Q:

$$f : Q \to \mathbb{R}$$
$$g : Q \to \mathbb{R}$$

where we define Q as the set of all possible paths joining v_{start} and v_{goal}. Further, let say we want to find a path $q \in Q$ such that it maximize the value of $f(q)$ under the constraint that $c \geq g(q)$.

In general, our problem can be formulated, by using Lagrange multipliers as :

$$J(P) = f(P) + \lambda(c - g(P))$$

where we want to maximize the *cost* $J(P)$. We can express this as a *minimization problem* by switching signs:

$$J(P) = -f(P) - \lambda(c - g(P)) \tag{5}$$

So now our goal is to minimize $J(P)$. For our specific problem, our goal is to find a sequence of paths such that the path P_{i+1} maximize the distance between itself and the previous paths. Formally speaking we can define $f(.)$ as:

$$f(P_{i+1}) = D_{added}(P_{i+1}, \mathscr{P}_i)$$

where

$$\mathscr{P}_i = \{P_1, ...P_i\}$$

and $g(P_i)$ is the cardinality of the path (representing its length):

$$g(P_{i+1}) = \mid P_{i+1} \mid$$

and c represents the upper bound of length required. Hence, we can rewrite Equation (5) for our particular problem as:

$$J(P_{i+1}) = -D_{added}(P_{i+1}, \mathscr{P}_i) - \lambda(c - \mid P_{i+1} \mid)$$

$$J(P_{i+1}) = -\sum_{k=1}^{|P_{i+1}|} D_{VS}(p_{i+1k}, \mathscr{P}_i) - \lambda(c - \mid P_{i+1} \mid)$$

$$J(P_{i+1}) = -\sum_{k=1}^{|P_{i+1}|} D_{VS}(p_{i+1k}, \mathscr{P}_i) + \mid P_{i+1} \mid D_{max} - \mid P_{i+1} \mid D_{max} - \lambda(c - \mid P_{i+1} \mid)$$

$$J(P_{i+1}) = \sum_{k=1}^{|P_{i+1}|} \{D_{max} - D_{VS}(p_{i+1k}, \mathscr{P}_i)\} - \mid P_{i+1} \mid D_{max} - \lambda(c - \mid P_{i+1} \mid)$$

$$J(P_{i+1}) = \sum_{k=1}^{|P_{i+1}|} \{D_{max} - D_{VS}(p_{i+1k}, \mathscr{P}_i)\} + \mid P_{i+1} \mid (\lambda - D_{max}) - c\lambda$$

c is a constant and λ is a parameter, so minimizing the expression above is equivalent to minimize:

$$J(P_{i+1}) = \sum_{k=1}^{|P_{i+1}|} \{D_{max} - D_{VS}(p_{i+1k}, \mathscr{P}_i)\} + (\lambda - D_{max}) \mid P_{i+1} \mid$$

if $(\lambda - D_{max}) = \alpha$:

$$J(P_{i+1}) = \sum_{k=1}^{|P_{i+1}|} \{D_{max} - D_{VS}(p_{i+1k}, \mathscr{P}_i)\} + \alpha \mid P_{i+1} \mid$$

this is the cost we aim to minimize. We further restrict:

$$\lambda > D_{max} \rightarrow \alpha > 0$$

to make sure that the cost of a path $J(P_{i+1})$ is always positive. Now, what we want is to get an expression for the cost of a path based on the cost of each individual node composing the path, since this would let us find the path based on graph-search techniques. We start by inserting the α factor inside the sum:

$$J(P_{i+1}) = \sum_{k=1}^{|P_{i+1}|} \{D_{max} - D_{VS}(p_{i+1k}, \mathscr{P}_i) + \alpha \times 1\}$$

$$J(P_{i+1}) = \sum_{k=1}^{|P_{i+1}|} \{(D_{max} - D_{VS}(p_{i+1k}, \mathscr{P}_i)) + \alpha \times 1\}$$

from what we can see that the cost of a path is made up of the sum of the cost of each node, in which, for *each node* $n \in P_{i+1}$, the *node cost* is represented by:

$$Cost(n) = D_{max} - D_{VS}(n, \mathscr{P}_i) + \alpha \times 1 \tag{6}$$

From Eq.(6), we can conclude that the cost of a node is the sum of two separate costs:

- **The distance cost**:
$$D_{max} - D_{VS}(n, \mathscr{P}_i)$$
which can be understood as a penalization of *how close* a node is to the set of existing paths $\mathscr{P}_i$
- **The length cost:**
$$\alpha \times 1$$

which represents the penalization for the length added per each node. α can be understood as the importance factor of the length constraint with respect to the overall minimization problem. In the results section we will discuss the effect that the variation of α has in the set of paths produced.

From the expressions above, it is easily seen that the costs of each node are always positive, hence we can apply a graph-search algorithm to find the paths, according to the cost expressions just defined. The algorithm is shown:

Algorithm 1: Find-Sequence-Paths($\mathscr{V}$, $\mathscr{E}$, k, v_{start}, v_{target})

```
𝒫 = ∅
ResetSearch()
for i ← 1 to k do
    P_i = FindPath(v_start, v_target)
    𝒫.add(P_i)
    ResetSearch()
    UpdateNodeValues(𝒫)
return 𝒫
```

Algorithm 2: `ResetSearch()`

```
forall v ∈ 𝒱 do
    v.costF ← ∞ ;
    v.costG ← ∞ ;
    v.costH ← ∞ ;
    v.status ← NO_VISITED ;
    v.value ← 0 ;
OpenSet ← ∅
```

Algorithm 3: `FindPath`(v_{start}, v_{target})

```
v_start.costG ← 0 ;
v_start.costH ← v.Heuristic(v_target) ;
v_start.costF ← v_start.costG + v_start.costH;
v_start.parent ← −1
PushOpenSet(v_start) ;
iter ← 0 ;
while iter < maxIter do
    v = PopOpenSet() ;
    if v == v_target then
        BacktracePath(x) ;
    v.status ← IN_CLOSED_SET ;
    forall v_n ∈ v.neighbors do
        if v_n.status == IN_CLOSED_SET then
            continue ;
        costG' ← v.costG+ EdgeCost(v, v_n) ×(v_n.value + α × 1) ;
        if v_n.status ≠ IN_OPEN_SET then
```