Solid Freeform Fabrication 2018: Proceedings of the 29th Annual International Solid Freeform Fabrication Symposium – An Additive Manufacturing Conference
Reviewed Paper

Reinforcement Learning for Generating Toolpaths in Additive Manufacturing

Steven Patrick*, Andrzej Nycz*, and Mark Noakes*

*Manufacturing Demonstration Facility, Oak Ridge National Laboratory, Knoxville, TN

I. Abstract

Generating toolpaths plays a key role in additive manufacturing processes. In the case of 3-Dimensional (3D) printing, these toolpaths are the paths the printhead will follow to fabricate a part in a layer-by-layer fashion. Most toolpath generators use nearest neighbor (NN), branch-and-bound, or linear programming algorithms to produce valid toolpaths. These algorithms often produce sub-optimal results or cannot handle large sets of traveling points. In this paper, the researchers at Oak Ridge National Laboratory's (ORNL) Manufacturing Demonstration Facility (MDF) propose using a machine learning (ML) approach called reinforcement learning (RL) to produce toolpaths for a print. RL is the process of two agents, the actor and the critic, learning how to maximize a score based upon the actions of the actor in a defined state space. In the context of 3D printing, the actor will learn how to find the optimal toolpath that reduces printhead lifts and global print time.

II. Introduction

In most 3D printing processes, the path the printhead takes is not simple. It can vary drastically between two different printed parts. However, the software that determines these paths must have the capabilities to handle all the varying cases. Many heuristics have been proposed like Nearest Neighbor (NN), Branch-And-Cut [1], and linear programming [2] algorithms. However, these heuristics often fall short of finding the optimal path or take far too long to compute for practical purposes. The most common form of sub-optimal path has line breaks in it. Line breaks are detrimental in many aspects. They cause the print time to increase, which in an industrial setting, is a loss of money. These line breaks are also the weak points within a part. The reason for this is a solid line has much stronger bonds holding it together than a broken one. All printers have a certain degree of error in their positioning precision. Whenever there is a print lift relative distances can be altered by this error. However, if the bead is continuous, the relative positioning is held constant for the layer.

A. 3D Printing

A large majority of 3D printing processes involve extrusion. This process involves heating a material until it is soft and then forcing the material out of a nozzle at a desired location. Additionally, extrusion usually has three different types of patterns within a part. They are classified as infill, inset, and skeletons. Infill refers to the print material that is inside of the final 3D part. Usually, this is a repeated pattern that does not follow the outer edges of the part, such as a honeycomb or raster pattern. Insets are the perimeter beads that outline the part. Skeletons are smaller beads that fill the spaces where the infill or insets could not reach.

[0]The manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

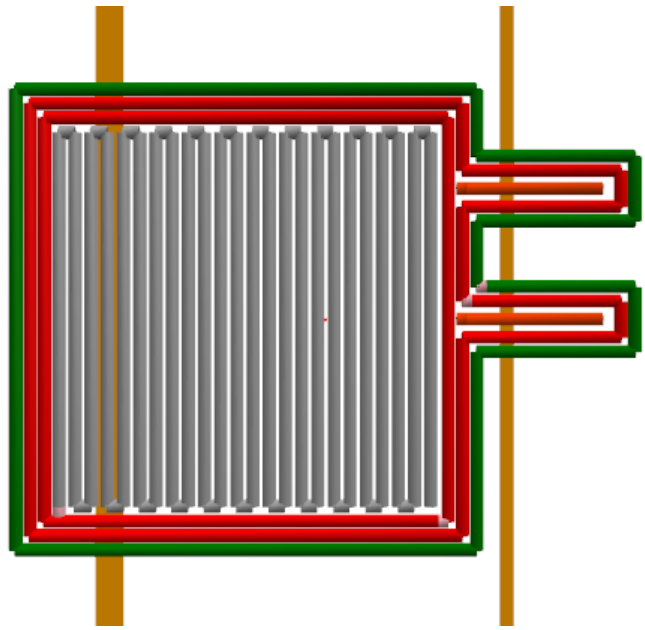


Fig. 1: Different Bead Types: Infill (Grey), Skeletons (Orange), Insets (Green and Red). The vertical brown tan lines are grid lines for the slicer print bed.

There are a multitude of different types of infill: parallel lines, grids, triangles, and hexagons. Examples of these infill patterns can be seen in Figure 2. All infill patterns have different strengths and weaknesses, but they all prove to be difficult to print in a single line. Therefore, the printhead needs to lift up in the z-axis and go to a different position to continue printing. This is known as a printhead lift, and it primarily occurs in the infill of a part.

Printhead lifts cause multiple problems for a print. An obvious issue is that the print time increases as the amount of printhead lifts increase. In an industrial setting, this means less product can be made. These defects are caused by a lack of smoothness at the beginning of a bead or a positioning precision error. Therefore, print lifts are not desired, and slicing software should try to minimize them in any way possible.

B. Traveling Salesman Problem

The traveling salesman problem (TSP) is a well-known issue in the realm of computer science. It refers to an agent that needs to go to multiple points on a map. A savvy salesman wants to visit as many cities with the least amount of travel possible. A solution may come to mind easily, but this problem scales factorially. The amount of calculations needed to try all possible options is n!, where n is the number of cities to travel to within a given map. This makes the TSP a non-deterministic polynomial time (NP-Hard) problem [3].

In 3D printing, this can range anywhere from 10 points to 1,000 points. The latter case would need to check more than 10^{249} options if brute force were applied. Even more concerning is that 10^{249} brute forced options could apply to just one layer. Almost all prints have more than a single layer; some have a layer count of over 100. Clearly brute force is not a valid option.

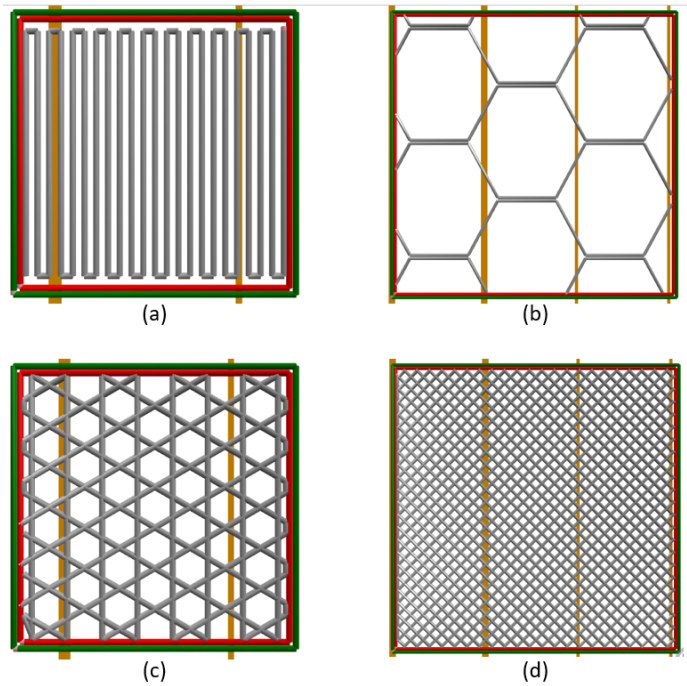


Fig. 2: Different Infill Patterns. a) Lines b) Hexagonal c) Triangle/Hexagon Hybrid d) Grid

C. Nearest Neighbor

There have been many approaches to solving the TSP. However, Nearest Neighbor (NN) has proven to be one of the fastest heuristics for solving the TSP. The NN algorithm works as follows: the computer reads its current state and assesses the distances between its current state and all of the valid cities to travel to within a given map. It then chooses the closest city to as its next destination. The pseudo code for this algorithm can be seen in Figure 3.

Fig. 3 Nearest Neighbor

```
1:  procedure NN
2:      Pos ← current position
3:      TravelTo ← points to travel to
4:      Visited ← points already visited
5:      if TravelTo = empty then
6:          goto 12
7:      j ← Index of closest point by Euc. distance within TravelTo with respect to Pos
8:      Add TravelTo[j] to the back of Visited
9:      Pos ← TravelTo[j]
10:     Remove TravelTo[j] from TravelTo
11:     goto 5.
12:     return Visited
```

NN is classified as a greedy algorithm which means it chooses the best option with regard to its current state. In other words, greedy algorithms do not take into consideration past actions or long-term future rewards. Therefore, large jumps may occur when the computer reaches the edge of the map. An example of NN performing poorly can be seen in Figure 4. As seen in this figure, the start point of the NN algorithm effects the overall travel distance drastically. On average, NN returns a travel plan that is 25% longer than the optimal path [4]. This is where a repetitive NN (RNN) comes in. It tries all the points as

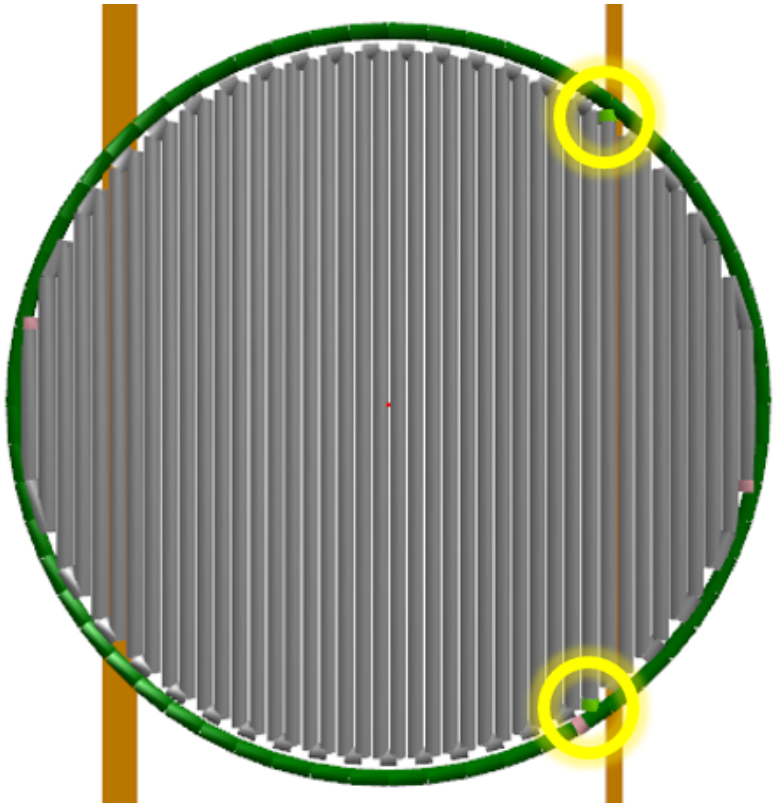


Fig. 4: Infill generation where NN does not work well. The bead starts (neon green points circled in yellow) a quarter past the center of the circle. Therefore, NN makes it impossible to do the infill in one bead

start points and uses the NN approach to plan a path. This improves the result, but it reduces the major benefit of NN. For a print with over 100 points to travel to, running NN can take a long time. The amount of distances needed to calculated can be found with this equation: $n\sum_{i=1}^{n} i = \frac{n^2(n+1)}{2}$. For the 100 points example, this would be equal to 505,000 calculations per layer. Most high-performance computers can handle this kind of computation, but it would take a couple of minutes to slice a part that had over 100 layers. Most people want to quickly slice parts in a matter of seconds.

D. Reinforcement Learning

An emerging field in machine learning (ML) is called reinforcement learning (RL). There have been many stories of major breakthroughs with this approach like playing games [5], robot locomotions [6], and energy efficiency [7]. There are many different types of implementations using RL, but they all have the same underlying principle. The actor will perform a task, it is then given a score by the critic. The actor then changes its policies and performs the task again. The critic gives a score again and the process continues on like that until the actor is performing to a desired standard.

III. METHODS

In order to improve the current standard, an RL algorithm was implemented. The underlying principle of this algorithm relies on the recurring nearest neighbor (RNN) approach mentioned in II-C. We offer a slight improvement to the RNN approach. Rather than checking all the travel points as start points, only points that cause a break in the print bead are considered. The reasoning for using the break points is because they tend to occur in corners or along edges of a part. Those are usually the places that are better to start printing rather than in the middle of a part.

To put this into RL terms, the actor first chooses a random point to start their path generation based upon the NN. The critic then counts the number of printhead lifts and where they occur and gives that score to the actor. The actor then takes these points and learns different strategies for creating an infill pattern. The actor then tries implementing NN on those points and gets a score. The actor finally looks at all the scores it received and chooses the path with the lowest associated score. The pseudo code can be seen in Figure 5.

Fig. 5 Reinforcement Learning Nearest Neighbor

```
1:  procedure NN
2:      Pos ← current position
3:      TravelTo ← points to travel to
4:      Visited ← points already visited
5:      numLifts ← 0
6:      liftPos ← Points where a break occurs, starts out empty
7:      if TravelTo = empty then
8:          goto 17
9:      j ← Index of closest point by Euc. distance within TravelTo with respect to Pos
10:     if TravelTo ← PostextCausesalift then
11:         numLifts++
12:         Add Pos, TravelTo[j] to liftPos
13:     Add to back TravelTo[j] to Visited
14:     Pos ← TravelTo[j]
15:     Remove TravelTo[j] from TravelTo
16:     goto 7.
17:     returnVisited, numLifts, liftPos
18: procedure RL
19:     Path, numLifts, Pos ← NN(Random Point)
20:     if numLifts! = 0 then
21:         j ← 0
22:         Add entry to Path, numLifts ← NN(Pos[j])
23:         j++
24:         if j < len(Pos) then
25:             goto 22
26:         else
27:             goto 28
28:     j ← Index of lowest value in numLifts
29:     returnPath[j]
```

IV. Results

As seen in Figure 6, the algorithm produced an infill pattern with a reduction in bead count. In every case the team tested, the new algorithm produced better or equivalent results than NN. Most cases had less than a five second addition in slicing time compared to the NN approach on an Intel Core i7-4790 CPU at 3.60 GHz and 8 GB of RAM.

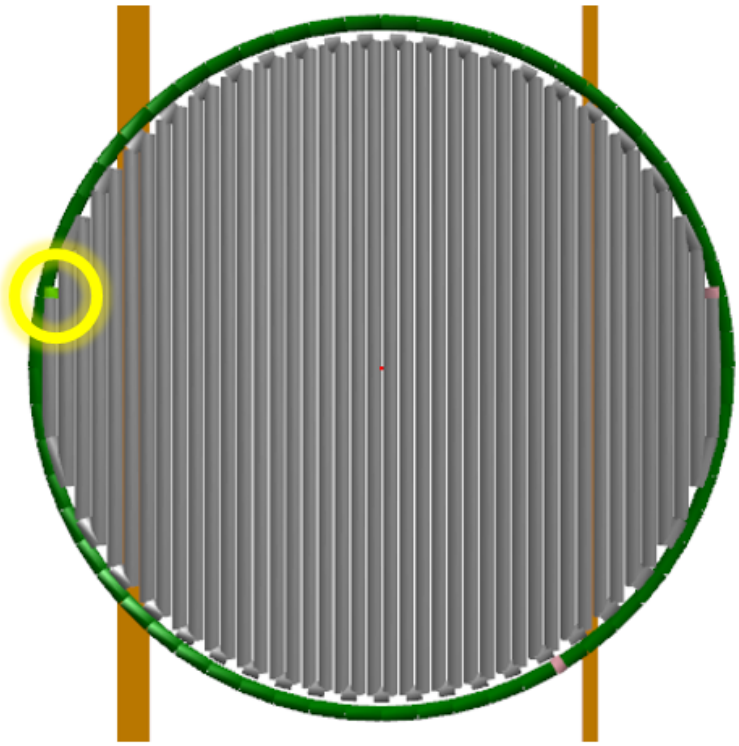


Fig. 6: Infill generation with the same part as Fig. 4. After learning where the break points are, the slicer knows to start as far away from the center as possible. This creates the entire infill in one bead as opposed to two beads in Figure 4

Another simple example of the benefits of the proposed RL algorithm can be seen in Figures 7 and 8. In this example, the figures show the improvement of the bead count from four beads to two. Therefore, this algorithm does not guarantee a continuous bead for every geometry, but it does guarantee that the resultant path will be less than or equal to the amount of beads from a NN approach.

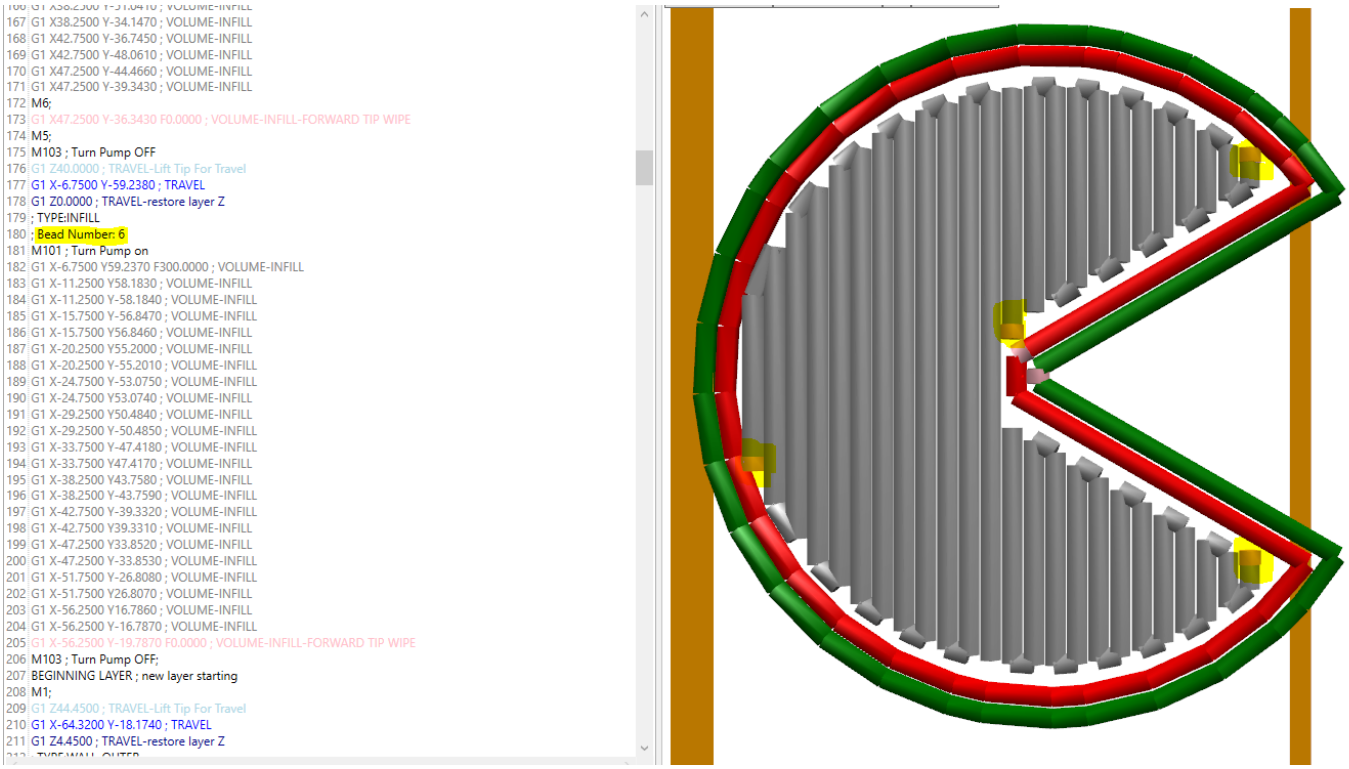


Fig. 7: Infill generation with ”Pacman”. 6 beads total (2 insets + 4 infill).

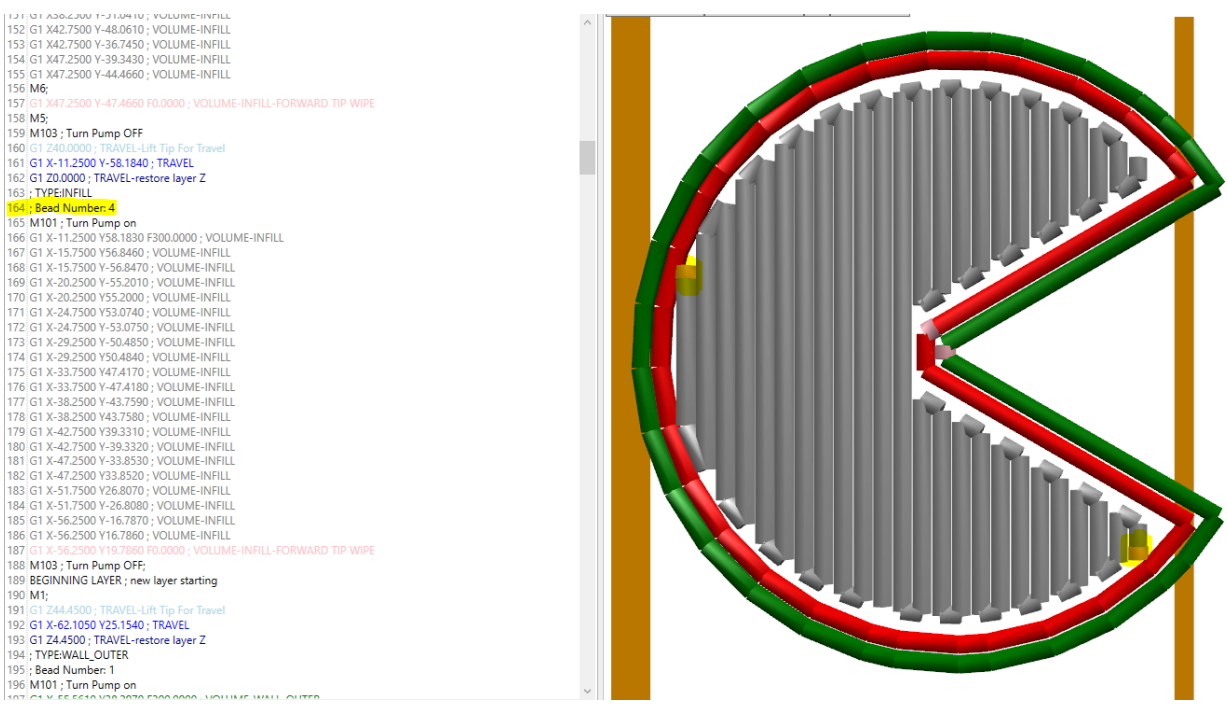


Fig. 8: Infill generation with the same part as Fig. 7. The total bead count is highlighted. 4 beads total (2 insets + 2 infill).

A more complicated geometry using both NN and RL can be seen in Figures 9 and 10 respectively. A highlight of this piece is it is not curved like the other geometries shown this far. The reason this layout is harder to print in a continuous line is that there are holes within the part. Just like in the previous example, there is a bead count reduction going from NN to RL. Additionally, this new case could not be printed in a single bead.

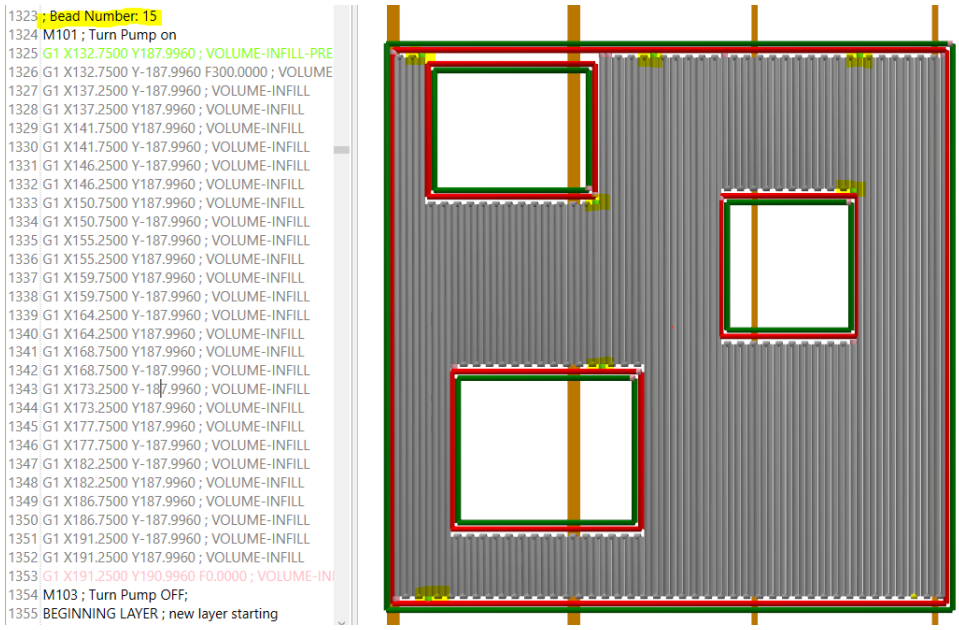


Fig. 9: Infill generation with a complicated geometry using NN. 15 beads total (8 insets + 7 infill). Neon green points indicate the start of a bead.

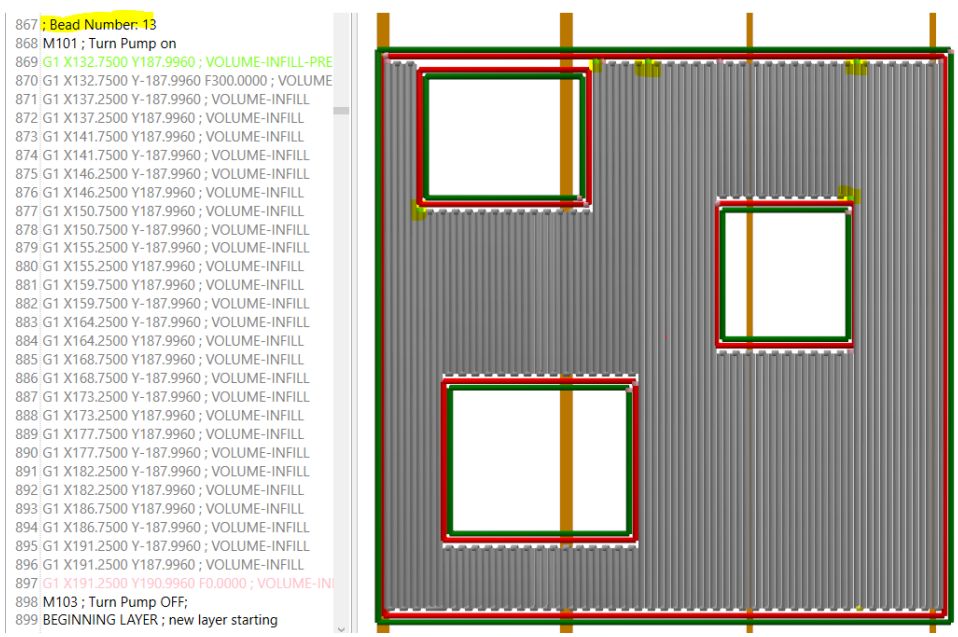
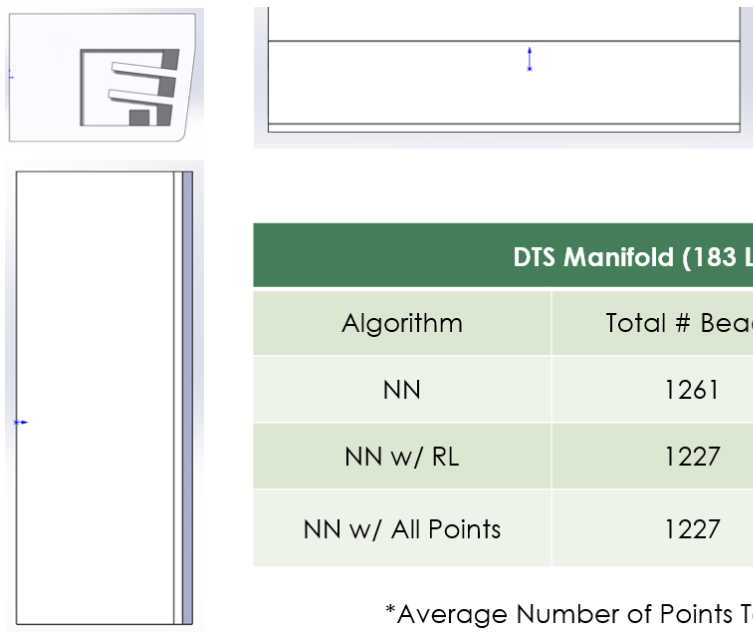


Fig. 10: Infill generation with the same part as Fig. 9, but using the proposed algorithm. The total bead count is highlighted. 13 beads total (8 insets + 5 infill). Neon green points indicate the start of a bead.

To illustrate the importance of this algorithm, the bead count for a part with a large layer count was analyzed. As seen in Figure 11, there is a significant bead count difference between the two slicing iterations. However, the added slicing time is negligible. Even though, a 2.7% difference in total bead count does not seem significant, it is actually quite large if you only consider the infill where the algorithm is applied. This algorithm does not get applied to insets or skeletons because these beads are not usually connected. Insets create a solid line perimeter and the skeletons fill in where the insets or infill could not reach. Since there was an average of six inset beads for every layer, there were roughly 1,098 insets that could not be optimized by the algorithm. With that taken into consideration, the infill bead count reduces by 21%.



DTS Manifold (183 Layers)		
Algorithm	Total # Beads	Slicing Time
NN	1261	20.1 s
NN w/ RL	1227	21.4 s
NN w/ All Points	1227	27.5 s

*Average Number of Points To Travel To Was 30

Fig. 11: Infill generation with a manifold. The orthographic views are shown on the left. The number of total beads in the part as well as the computation time for NN, NN with RL, and NN using all of the points as start points are shown in the table

In a few cases, the algorithm took a longer time because the number of bead breaks was large. These cases usually had many isolated beads that could not be combined with others. Therefore, the algorithm tried all these paths even though none of them could be combined to make a single bead. This means there would be no improvement upon the NN algorithm suggested path. An example of this type of failure can be seen in Figure 12.

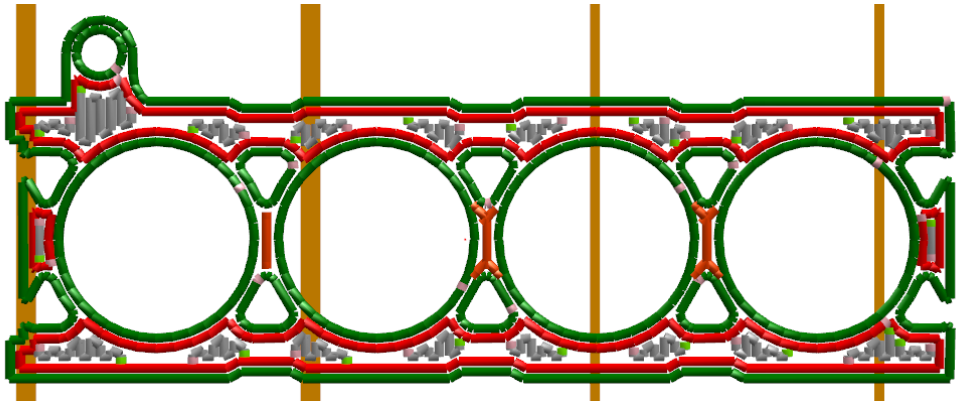


Fig. 12: Infill generation using RL. This part had many isolated beads that could not be connected. Therefore, it tried all the different options, but that extra computation time resulted in no improvement.

V. Conclusion

In all tested cases, RL produces an infill path that has fewer or equivalent number of beads when compared to NN. This is because the algorithm uses NN as its base and only chooses paths that are better than NN. The computation time added to the slicing process, on average, is less than five seconds on a Intel Core i7-4790 CPU at 3.60 GHz and 8 GB of RAM. If the print has a lot of initial lifts, then the algorithm takes longer to execute. From our tests, the additional computation time is more than compensated for by the reduction in bead count in almost all cases. It was shown that this approach performs best on parts with solid convex infill like in the first example shown. It still performs well with more complicated geometries like with the ”Pacman” shape and the part with holes in it, but there is no guarantee that the algorithm will produce an infill path that is a single continuous bead. Finally, the algorithm does not perform well in cases where there are multiple regions for the infill that are not connected as seen in the last example. Therefore, for parts with multiple isolated infill regions, RL should not be used.

References

[1] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[2] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[3] Donald Ervin Knuth. Postscript about np-hard problems. *ACM SIGACT News*, 6(2):15–16, 1974.

[4] David S Johnson and Lyle A McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1:215–310, 1997.

[5] Satinder Singh, Andy Okun, and Andrew Jackson. Learning to play go from scratch. *Nature*, 550:336 EP –, Oct 2017.

[6] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[7] Jim Gao. Machine learning applications for data center optimization, 2014.