The Thirteenth International Symposium
on Combinatorial Search (SoCS 2020)

Toward a String-Pulling Approach to Path Smoothing on Grid Graphs

Jihee Han, Tansel Uras, Sven Koenig

Department of Computer Science
University of Southern California
Los Angeles, USA
jrjrhan@gmail.com turas@usc.edu skoenig@usc.edu

Abstract

Paths found on grid graphs are often unrealistic looking in the continuous environment that the grid graph represents and often need to be smoothed after a search. The well-known algorithm for path smoothing is greedy in nature and does not necessarily eliminate all heading changes in freespace. We present preliminary research toward a new path-smoothing algorithm based on "string pulling" and show experimentally that it consistently finds shorter paths than the greedy path-smoothing algorithm and produces paths with no heading changes in freespace.

Introduction

Grid graphs are useful discretizations that allow for efficient path planning in continuous 2D environments via A* and other graph-search algorithms. Grid graphs are *angle-limited*, that is, allow movement in only a fixed number of directions. As a result, shortest grid paths are often longer than shortest paths in the continuous environments and unrealistic looking due to unnecessary and thus unmotivated heading changes in freespace. This problem can be mitigated by *smoothing* grid paths in a post-processing step, typically with a greedy path-smoothing algorithm that replaces parts of the paths with straight lines that do not intersect with obstacles (Botea, Müller, & Schaeffer 2004; Thorpe 1984; Millington & Funge 2009). It can also be mitigated by interleaving the path smoothing with the search, resulting in any-angle path-planning algorithms (Ferguson & Stentz 2006; Nash & Koenig 2013; Sislak, Volf, & Pechoucek 2009; Choi, Lee, & Yu 2010; Yap, Burch, Holte, & Schaeffer 2011; Harabor & Grastien 2013; Uras & Koenig 2015). Any-angle path-planning algorithms are typically slower than A* followed by path smoothing, but find shorter paths.

In this paper, we introduce a path-smoothing algorithm based on "*string pulling*:" Imagine that our input path (a shortest grid path) is a piece of string. If we pull the string taut between the start and goal vertices, then the resulting configuration of the string corresponds to a shortest (any-angle) path between the start and goal vertices in the continuous environment that is in the same homotopy class as the input path (that is, circumnavigates the obstacles in the same way). Our work is research in progress, and we currently do not have a proof that our string-pulling algorithm perfectly simulates string pulling and therefore produces the shortest (any-angle) path in a given homotopy class. However, we show experimentally that it consistently finds shorter paths than the greedy path-smoothing algorithm and produces paths with no heading changes in freespace.

This paper is organized as follows. We first describe the existing greedy path-smoothing algorithm and highlight a fundamental weakness of this algorithm, namely that its paths can have heading changes only at vertices along the input path and thus have heading changes in freespace for some input paths. We then introduce our string-pulling algorithm and describe how it can efficiently identify new vertices for heading changes. We conclude by presenting experimental results comparing the two path-smoothing algorithms and Theta*, a prototypical any-angle path-planning algorithm (Daniel, Nash, Koenig, & Felner 2010).

Preliminaries

A grid is a tessellation of a 2D environment into square cells, where each cell is either blocked or unblocked. An eight-neighbor grid graph $G = (V, E)$ is constructed from the grid by placing vertices at the corners of unblocked cells and connecting two vertices with an edge iff they belong to the

same unblocked cell. We use *s* and *g* to denote the start and goal vertices, respectively. Two vertices *u* and *v* have line-of-sight (LOS) iff the straight-line segment between them is contained in the union of unblocked cells (including their borders). Whether two vertices have LOS can be determined with a modified version of Bresenham's line-drawing algorithm (Bresenham 1965), as described in (Botea, Müller, & Schaeffer 2004).

A path is a sequence of vertices. An any-angle path is a sequence of vertices where consecutive vertices on the path have LOS. A grid path is an any-angle path where consecutive vertices on the path are neighbors. The input path *P* is the grid path ($s = p_1, p_2, \ldots, p_n = g$). According to our definition of LOS, any-angle or grid paths can pass through the vertex were diagonally-touching blocked cells touch, an assumption that we make only for simplicity.

Greedy Path-Smoothing Algorithm

Removing an internal vertex p_i from an any-angle path *P* (that is, a vertex on the path that is not the first or last vertex on the path) produces another any-angle path iff the immediately preceding and succeeding vertices p_{i-1} and p_{i+1} of p_i on *P* have LOS. Removing p_i from *P* shortcuts p_i and makes the path move directly from p_{i-1} to p_{i+1} on a straight line. The greedy path-smoothing algorithm operates by iterating over the internal vertices of the input path and shortcutting them if possible (Botea, Müller, & Schaeffer 2004). Figure 1 shows an example, where the input path is shown in grey, the output path is shown in green, and the shortest any-angle path is shown in blue. The greedy path-smoothing algorithm removes the first three internal vertices of the input path after successful LOS checks. It does not remove the fourth internal vertex after an unsuccessful LOS check, but removes the fifth internal vertex again, which does not shorten the path any further. The resulting path is longer than the shortest any-angle path and has an unmotivated heading change in freespace at the fourth internal vertex of the input path.

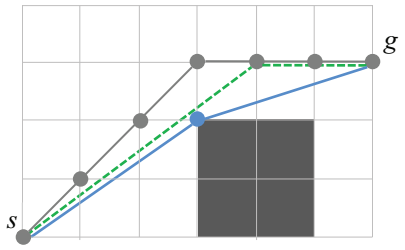

Figure 1: Example of the greedy path-smoothing algorithm

If the greedy path-smoothing algorithm had iterated over the internal vertices in a different order, it could have produced a different output path. For example, if it had started at the fourth internal vertex, it would have successfully shortened it. However, its output path is always restricted to be a subsequence of its input path since it only removes vertices but does not add them. Therefore, the heading changes of its output path can occur only at the internal vertices of its input path, which is why it cannot produce the shortest any-angle path in Figure 1, regardless of the order in which it iterates over the internal vertices of its input path.

String-Pulling Algorithm

Our string-pulling algorithm aims to produce a shortest any-angle path in the same homotopy class as the input path by both removing vertices from the input path and, different from the greedy path-smoothing algorithm, adding vertices to it. It starts with the empty output path and iterates over the vertices on the input path in the order in which they appear on it. We use *SP* to denote the output path and sp_{end} and sp_{end-1} to denote the last and next to last, respectively, vertex on it. During each iteration, it either adds a vertex to *SP* or removes a vertex from it. It adds ("appends") a new vertex to the end of *SP* to add a necessary heading change to it if a LOS check fails, based on the blocked cells that cause the LOS check to fail. It removes ("truncates") the last vertex from *SP* to remove the heading change from it if that heading change is no longer a taut turn at a convex corner of a blocked cell. During each iteration, it can append or truncate any number of vertices (including no vertices). Algorithm 1 shows pseudocode of the string-pulling algorithm, and Table 1 shows a trace of its operations for the example of Figure 2. We now explain its operations in more detail.

Adding a vertex to *SP*: The key insight behind adding a vertex to *SP* is that, when a LOS check fails, we can determine which vertex to append based only on the set of blocked cells that caused the LOS check to fail. For instance, in the example of Figure 1, once the LOS check to shortcut the fourth internal vertex on the input path fails, instead of keeping that vertex on the output path, we replace it with the blue vertex at the corner of the blocked cell that caused the LOS check to fail. We now provide details on the approach.

We consider the last vertex sp_{end} on the current output path *SP* (initially p_1) and vertex p_i (initially p_3). If the LOS check between sp_{end} and p_i fails (Lines 7-18), we determine the set *icell* of cells that the straight line between sp_{end} and p_i intersects (by invoking function *LOScells* on Line 5). We then determine the set *cd* of candidate vertices to be appended to *SP* as all corners of all cells in *icell* (by invoking function *GetCorners* on Line 7). For each candidate vertex *v'* in *cd*, we determine the angle $angle_{v'} = \angle(p_{i-1}, sp_{end}, v')$ (by invoking function *CalcAngle* on Line 9), append the one with the smallest such angle (Lines 8-11) to *SP* as the vertex that results in the most taut collision-free

path between sp_{end} and p_i, and iterate. For example, in Figure 2(b), the LOS check fails due to a blocked cell, whose corners become candidate vertices to be appended to *SP* (shown as red circles), among which we choose the one (namely, B3) with the smallest angle (namely, zero). In case of ties, we choose the one farthest away from sp_{end} (Lines 12-15) to keep the number of iterations small. In Figure 2(d), two candidate vertices (namely, B3 and A5) have the smallest angle (namely, zero), and we thus choose the one (namely, A5) farthest away from sp_{end} (namely, C1). If we chose B3 instead, then we would need an additional iteration to append A5 and the resulting path would contain the unnecessary vertex B3 since the path passes through it in a straight line.

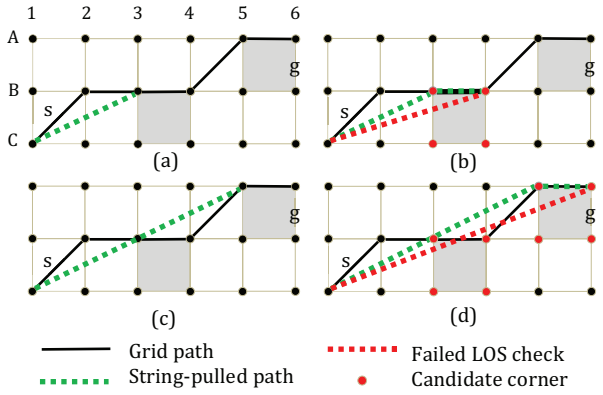

Figure 2: Example of the string-pulling algorithm

Iteration	Figure	Lines	*SP*	*turn*
in the beginning	2(a)	1-3	(C1)	NULL
i=3; C1-B3 have LOS: truncate	2(a)	20-29	no change since *size*(*SP*) = 1	
i=4; C1-B4 have no LOS: append	2(b)	7-18	(C1, B3)	*CalcTurn*(C1, B3, B4) = right
i=4; B3-B4 have LOS: truncate	2(b)	20-29	no change since turn has not changed	
i=5; B3-B5 have LOS: truncate	2(c)	20-29	(C1)	no change since *size*(*SP*) = 1
i=5; C1-A5 have LOS: truncate	2(c)	20-29	no change since *size*(*SP*) = 1	
i=6; C1-A6 have no LOS: append	2(d)	7-18	(C1, A5)	*CalcTurn*(C1, A5, A6) = right
i=6; A5-A6 have LOS: truncate	2(d)	20-29	no change since turn has not changed	
at the end	2(d)	32-33	(C1, A5, A6)	no change

Table 1: Trace of the string-pulling algorithm

Removing a vertex from *SP*: The key insight behind removing a vertex from *SP* is that, when *SP* starts to turn in a different direction, it is no longer taut and we have to remove the last vertex from it. We now provide details on the approach.

If the LOS check between sp_{end} and p_i succeeds (Lines 20-29), we check whether *SP* is still taut. If not, then we truncate the last vertex on *SP* to make *SP* taut again and iterate. The following iteration will handle the case where the LOS check between the last vertex on *SP* after the update and p_i fails (by adding a vertex to *SP*). We determine whether *SP* is still taut by using the variable *turn* (initialized with NULL) to keep track of the kind of turn (left, straight, or right) from the next to last vertex on *SP* via the last vertex on *SP* to p_i (by invoking function *CalcTurn*). Whenever a vertex is added to or removed from *SP*, we update the value of variable *turn* (Lines 17 and 25). If the LOS check between sp_{end} and p_i succeeds, then we calculate the current kind of turn (Line 21) and compare it to the one stored in variable *turn*. If the two values are different, then *SP* is no longer taut and we truncate the last vertex on *SP* to make *SP* taut again (Lines 20-29). In Figure 2(b), B3 is appended and the resulting turn of (C1, B3, B4) is to the right. In Figure 2(c), B3 is truncated because the turn of (C1, B3, C5) is straight and thus different from the previous one.

Algorithm 1 String-Pulling ($P = (p_1, p_2, \ldots, p_n)$)

```
1:  SP ← ∅;
2:  SP.append(p_1);
3:  turn ← NULL;
4:  for i = 3, …, n
5:      icell ← LOScells(sp_end, p_i);
6:      if icell ≠ ∅
7:          cd ← GetCorners(icell);
8:          for each v' ∈ cd
9:              angle_v' ← CalcAngle(p_{i-1}, sp_end, v');
10:         end
11:         cd^min ← {v ∈ cd | angle_v = min_{v'∈cd} angle_v'};
12:         for each v' ∈ cd^min
13:             dist_v' ← ||sp_end − v'||;
14:         end
15:         u ← argmax_{v'∈cd^min} dist_v';
16:         SP.append(u);
17:         turn ← CalcTurn(sp_{end-1}, sp_end, p_i);
18:         i ← i − 1;
19:     else
20:         if size(SP) > 1
21:             cur_turn ← CalcTurn(sp_{end-1}, sp_end, p_i);
22:             if turn ≠ cur_turn
23:                 SP.truncate();
24:                 if size(SP) > 1
25:                     turn ← CalcTurn(sp_{end-1}, sp_end, p_i);
26:                 end
27:                 i ← i − 1;
28:             end
29:         end
30:     end
31: end
32: SP.append(p_n);
33: return SP;
```

Experiments

We used grid maps of size 512x512 from Nathan Sturtevant's repository (http://movingai.com/benchmarks/) in our

experiments, including randomly blocked maps with different percentages of blocked cells, real-world street maps with different shapes of obstacles, and room maps with different room sizes. We ran our experiments on an Intel i5-6300U (2.40GHz) CPU with 4GB of RAM and compared A* without path smoothing, A* with the greedy path-smoothing algorithm (A* with G), our A* with string pulling (A* with SP), and Theta*, all implemented in C++. (We implemented Theta* both without path smoothing and with the greedy path-smoothing algorithm, resulting in similar statistics, so we report results for Theta* without path smoothing here.) Figure 3 shows the paths of A* without path smoothing, A* with G, A* with SP, and Theta*, which are drawn with a thin grey line, dotted green line, solid blue line, and dashed black line, respectively.

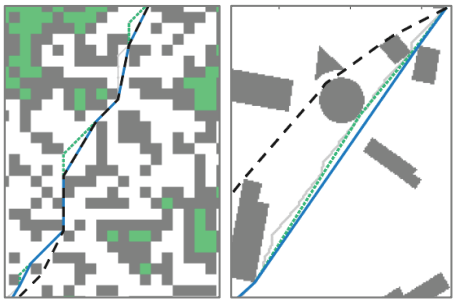

Figure 3: Example paths of different path-planning algorithms

		Path length (% longer than shortest)				Runtime (ms)				Freespace heading changes			
Map	No. of instances	A*	A* with G	A* with SP	Theta*	A*	A* with G	A* with SP	Theta*	A*	A* with G	A* with SP	Theta*
Rand 10	1,780	4.39	1.90	1.26	0.15	51	51	53	64	34.68	7.53	0.00	1.6
Rand 20	1,910	4.26	2.27	1.26	0.21	49	49	51	99	24.28	7.66	0.00	1.32
Rand 30	2,070	4.29	2.45	1.15	0.24	51	51	53	135	20.88	8.43	0.00	1.15
Rand 40	3,170	4.29	2.20	0.85	0.25	30	30	31	114	20.88	8.20	0.00	1.45
Str Berlin	1,870	5.08	0.82	0.13	0.08	84	85	86	478	37.48	1.31	0.00	0.25
Str Boston	1,830	4.42	0.72	0.30	0.08	61	61	62	339	34.84	1.46	0.00	0.39
Str NewYork	1,820	4.86	0.76	0.07	0.11	72	72	74	296	34.56	1.86	0.00	0.56
Str Paris	1,900	4.90	1.06	0.27	0.12	81	81	82	354	35.32	2.44	0.00	0.76
Room 8	2,140	4.91	1.55	0.12	0.16	111	111	113	297	40.50	11.22	0.00	1.87
Room 16	2,010	4.88	1.17	0.11	0.09	123	123	124	308	25.94	5.40	0.00	0.78
Room 32	2,130	5.26	0.98	0.07	0.05	149	149	150	449	20.79	2.89	0.00	0.21
Room 64	2,150	5.22	0.57	0.02	0.01	158	159	160	694	20.84	1.25	0.00	0.09

Table 2: Experimental results

Table 2 shows the path length, runtime, and number of heading changes in freespace for each map, averaged over all instances with random start and goal vertices.

Path length: The path-planning algorithm with the shortest path length was Theta* (with an average optimality gap of 0.13%), followed by A* with SP (0.47%), A* with G (1.37%), and A* without path smoothing (4.73%). We determined the optimality gaps by finding the shortest paths in the continuous environments with ANYA (Harabor & Grastien 2013).

Runtime: The path-planning algorithm with the shortest runtime was A* without path smoothing (with an average runtime of 173.17ms), followed by A* with G (173.69ms), A* with SP (176.71ms), and Theta* (302.74ms) – the opposite of the ordering of the path-planning algorithms according to their path lengths (as expected since a longer runtime should result in shorter paths).

Freespace heading changes: The path-planning algorithm with the least number of heading changes in freespace was A* with SP (with an average number of zero heading changes in freespace), followed by Theta* (0.86), A* with G (4.97), and A* without path smoothing (29.25).

Conclusions

We introduced a new easy-to-implement path-smoothing algorithm based on "string pulling" and showed experimentally that it consistently finds paths that are shorter than the ones of the greedy path-smoothing algorithm although it runs almost as fast as it. In fact, we showed experimentally that it finds paths that are almost as short as the ones of Theta* but runs much faster than it. Finally, we showed experimentally that it finds paths with fewer heading changes in freespace than the ones of both the greedy path-smoothing algorithm and Theta*, namely paths with no heading changes in freespace (that is, all heading changes help to circumnavigate obstacles via taut turns at the convex corners of blocked cells).

Our string-pulling algorithm is especially useful for applications where any-angle path-planning algorithms like Theta* are impractical to implement or use (for example, run too slowly) or unmotivated heading changes in freespace need to be avoided (for example, because they look unrealistic). It is also useful if an any-time algorithm is needed since it decreases the length of the input path over time. Finally, it is also useful if one wants to minimize the sum of the times for path planning and path following since it shortens larger and larger prefixes of the input path and an agent can thus start to move along the path during path smoothing.

Future work includes proving the correctness of the string-pulling algorithm (which might include changing it appropriately for this purpose), extending it to higher dimensions, and combining it with Theta*.

Acknowledgments

This research was performed while Jihee Han visited the University of Southern California, partially supported by a

grant of the National Research Foundation (NRF) of Korea funded by the Korean government (MSIT) under grant number NRF-2019R1C1C1002798. The research was also supported by grants funded by the National Science Foundation (NSF) of the US under grant numbers 1724392, 1409987, 1817189, 1837779, and 1935712.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1(1): 7-28.

Bresenham, J. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal* 4(1): 25–30.

Choi, S.; Lee, J.; and Yu, W. 2010. Fast any-angle path planning on grid maps with non-collision pruning. In *Proceedings of the IEEE International Conference on Robotics and Biomimetics*, 1051-1056.

Daniel, K.; Nash, A.; Koenig, A.; and Felner, A. 2010. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research* 39: 533-579.

Ferguson, D. and Stentz, A. 2006. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, 23(2): 79-101.

Harabor, D. and Grastien, A. 2013. An optimal any-angle pathfinding algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 308–311.

Millington, I. and Funge, J. 2009. *Artificial Intelligence for Games*. Morgan Kaufmann, second edition.

Nash, A. and Koenig, S. 2013. Any-angle path planning. *AI Magazine* 34(4): 85-107.

Sislak, D.; Volf, P.; and Pechoucek, M. 2009. Accelerated A* path planning. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 1133–1134.

Thorpe, C. 1984. Path relaxation: Path planning for a mobile robot. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 318–321.

Uras, T. and Koenig, S. 2015. An empirical comparison of any-angle path-planning algorithms. In *Proceedings of the Annual Symposium on Combinatorial Search*.

Yap, P.; Burch, N.; Holte, R.; and Schaeffer, J. 2011. Any-angle path planning for computer games. In *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*.