Multi-Target Search in Euclidean Space with Ray Shooting

Ryan Hechenberger, Daniel D. Harabor, Muhammad Aamir Cheema, Peter J. Stuckey, Pierre Le Bodic
Faculty of Information Technology, Monash University, Australia
{ryan.hechenberger, daniel.harabor, aamir.cheema, peter.stuckey, pierre.lebodic}@monash.edu

Abstract

The shortest path problem (SPP) asks us to find a minimum length path between two points, usually on a graph. In a Euclidean environment the points are in a 2D plane and here the path must avoid a set of polygonal obstacles. Solution methods for this *Euclidean SPP* (ESPP) typically convert the continuous 2D map into a discretised representation, like a graph or navigation mesh. RayScan is a recent and fast ESPP algorithm which avoids the preprocessing step by using a combination of "ray shooting" and polygon scanning. In this paper we improve the performance of RayScan using spatial reasoning and ray caching techniques. We also extend the algorithm, from single-target search to a multi-target setting. Comparative game map experiments show a substantial speedup.

Introduction

ESPP is important for applications such as video games (Algfoor, Sunar, and Kolivand 2015) where the input given is not a graph but a set of *obstacles* to avoid. These settings are often dynamic, in the sense that obstacles can be moved, added or removed between start-target queries. Conventional methods for solving ESPP include the Visibility Graphs (VG) (Lozano-Pérez and Wesley 1979) and Navigation Meshes (Cui, Harabor, and Grastien 2017), both of which use preprocessing to convert the Euclidean map to a discrete form more suitable for search. One issue is that every time the environment changes the discrete representation must be updated and these repair costs must be paid online.

Recently we developed RayScan (Hechenberger et al. 2020), a new ESPP algorithm without any preprocessing or repair costs. RayScan proceeds in the manner of an A* search (Hart, Nilsson, and Raphael 1968) and can be understood as discovering on-the-fly the edges of a Sparse VG (Oh and Leong 2017) by performing ray shooting operations. In this paper we describe four performance improving enhancements for RayScan: *block, skip, bypass* and *cache*, which together help to reduce the number of ray shooting operations and which can substantially improve RayScan's performance. We also consider multi-target settings, common in video games, where one needs to find a Euclidean shortest path from a single starting point to a set of targets t_i simultaneously (cf. running separate start to target queries).

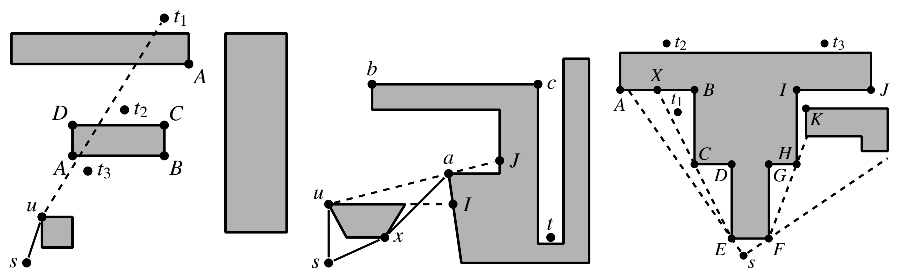
RayScan

RayScan's core concept is *ray shooting*, where we shoot (equiv. *cast*) a ray from some point u in a direction to determine the first intersecting obstacle and the point of intersection. When shooting from u to v there may be no obstacles between the two points. Then we have discovered an edge of the VG and consequently v is a successor of u in the search. Alternatively, we can discover an impeding obstacle between u and v and must now consider how to get around: either clockwise (CW) or counter-clockwise (CCW). After finding the obstacle, we perform a *scan*. We scan by rotating a *scan-line*: initially from the expanding node u to the intersection point and then rotating around u, following the edges of the obstacle in a specified orientation – CW or CCW. The scanning process stops when we find a type of vertex called a *turning point*. The turning point is designated v and results in a new ray being shot, from u to v. The entire shoot-and-scan procedure is implemented recursively and it guarantees to eventually discover an edge of the VG.

With recursive algorithms, we need a base case to prevent infinite looping. This is done with two methods. The first ends the recursion when a ray that we have already shot is proposed to be shot again. The second is by means of an *angled sector* (AS). The AS has an extreme CW and CCW angle, where any angle from the CW angle turning CCW up to the CCW angle is considered to be inside the AS, otherwise it is outside. Each scan is given an AS that it must stay within, otherwise the recursion ends; we can then split the AS into the CW and CCW parts defined from a new ray shot. These are given as the new AS' in the next recursive steps.

Blocking Extension

Blocking is an extension of RayScan that avoids redundant shooting, especially in multi-target settings. Consider Figure 1a: when expanding s we first shoot to t_1, which is blocked by obstacle $ABCD$. When scanning CW from that ray, the scan-line follows the edges until it reaches turning point B. Notice the scan-line passes over t_2 which is blocked by edge AB. Meanwhile target t_3 is not blocked. Once the t_1 scan is completed, we skip shooting to t_2 and shoot to t_3. The blocking tests are extremely cheap compared to ray shooting and as more targets are added the savings magnify.



(a) Blocking extension (b) Skip extension (c) Bypass extension

Figure 1: Auxiliary diagrams

Skip Extension

The skip extension avoids shooting to a vertex whose path length is worse than we already discovered. When expanding u, we consider turning point v. Normally we shoot a ray from u to v and continue. However if the g value of u plus distance from u to v is greater than the g value of v then uv cannot belong to any optimal path. Skip will avoid shooting the uv ray by trying to *scan-past* the turning point v by continuing the scan, ignoring the orientation of the scan line until that line passes the uv ray, in which case we skip shooting that ray and continue to the next turning point.

Take Figure 1b for example, where expanding u, our scan CCW goes from I to turning point a. Path $s - u - a$ is longer than $s - x - a$, thus we do not need to consider edge ua, and will attempt to skip the ray shot by continuing the scan, ignoring the CCW orientation requirements. Since this scan does succeed at passing the ua ray when it passes point J, the skip succeeds and we continue to find turning point b. If we were unable to skip a turning point, we will resort to shooting a ray and progress to the next scan(s).

Bypass Extension

The bypass extension avoids shooting rays from expanding vertex u to vertex v in cases where v leads to a dead end. We explain by example with Figure 1c. Let $u = E$ and $v = C$. We consider C for bypass by attempting to scan-past C; in this case passing the EC ray at point X. Being able to scan-past C is the first requirement of bypass. The second is that the scan-past *must* stay within the scan's AS. The third is dependent on the target's location: if the scan-past scan-line intersects the target then we cannot bypass, if it does not then the third condition is met. For this example, if t_1 is our target, then the scan-line intersects t_1 and we cannot bypass C. Meanwhile if t_2 is our target then bypass is possible; we avoid a ray shot from C and continue scanning from X.

This leads to the possibility of considering the same vertex twice, where first we bypass but second time we do not. For example, let $t = t_3$, $u = F$ and $v = H$, the first time reaching H is by a CW scan from FH. Here bypass succeeds and we find turning point J, leading into a CCW scan from ray FJ to find turning point K, followed by a CW scan from ray FK to again find H. This time when considering H for bypass the scan-past will leave the AS and thus fail. In this case we shoot FH and generate $v = H$ as a successor.

Single Target (µs)				**Multi Target** (ms)		
Algs	**S1**	**S2**	**S3**	**Algs**	**M1**	**M2**
R	400	2548	2689	R	18.83	321
RB	388	2492	2627	RB	**16.02**	**99.42**
RS	375	2392	2090	-	-	-
RP	262	1638	1859	-	-	-
R+	**248**	**1560**	**1395**	-	-	-
RC	150	736	683	RC	8.53	235.1
R+C	**132**	**723**	**625**	RBC	**5.634**	**48.56**

Table 1: Average runtime comparison of extensions for Starcraft 2 maps; **S1** Aftershock; **S2** Aurora; **S3** ArcticStation; **M1** Aurora (5,20,50 targets); **M2** Aurora (500,1000,2000 targets); Algs lists extensions: **R** RayScan; **B** blocking; **S** skip; **P** bypass; **R+** RBSP; **C** ray caching

Caching Extension

Caching is an extension that aims to speed up ray shooting by storing the results of previously shot rays (excluding rays from the start or to the target). The cache works by exploiting the fact that the search space of previous queries often overlap with the search space of the current query. Implementing the cache requires additional memory and its contents are invalidated each time the environment changes.

Results

Table 1 shows an ablation study of our extension methods (source code[1]). We experiment on maps from the well known StarCraft benchmark set (Sturtevant 2012). We see the Blocking extension has large speedups for multi-target queries. Skip and Bypass help, while Caching has the most significant impact. In experiment M2 blocking helps more than caching as most rays shot are towards targets and therefore not in the cache. Our full paper (Hechenberger et al. 2021) has more details and experiments and includes comparisons with leading mesh-based planners.

Acknowledgements

This work is supported by the Australian Research Council (ARC) under grant numbers DP190100013, DP200100025 and FT180100140 as well as a gift from Amazon.

[1]https://bitbucket.org/ryanhech/rayscan/

References

Algfoor, Z. A.; Sunar, M. S.; and Kolivand, H. 2015. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology* 2015: 1–11. ISSN 1687-7047.

Cui, M. L.; Harabor, D.; and Grastien, A. 2017. Compromise-free Pathfinding on a Navigation Mesh. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 496–502. AAAI Press.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2): 100–107. ISSN 0536-1567.

Hechenberger, R.; Harabor, D.; Cheema, M. A.; Stuckey, P. J.; and Bodic, P. L. 2021. Multi-Target Search in Euclidean Space with Ray Shooting (Full Version). URL https://people.eng.unimelb.edu.au/pstuckey/papers/rayscanm.pdf. arXiv preprint 2021.

Hechenberger, R.; Stuckey, P. J.; Harabor, D.; Le Bodic, P.; and Cheema, M. A. 2020. Online Computation of Euclidean Shortest Paths in Two Dimensions. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling*, 134–142. AAAI Press.

Lozano-Pérez, T.; and Wesley, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM* 22(10): 560–570. ISSN 0001-0782.

Oh, S.; and Leong, H. W. 2017. Edge N-Level Sparse Visibility Graphs: Fast Optimal Any-Angle Pathfinding Using Hierarchical Taut Paths. In *Proceedings of the Tenth International Symposium on Combinatorial Search (SoCS 2017)*.

Sturtevant, N. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2): 144 – 148. URL http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.