Dual Euclidean Shortest Path Search (Extended Abstract)

Ryan Hechenberger, Peter J. Stuckey, Pierre Le Bodic and Daniel D. Harabor

Faculty of Information Technology, Monash University, Australia
{ryan.hechenberger, daniel.harabor, pierre.lebodic, peter.stuckey}@monash.edu

Abstract
The Euclidean Shortest Path Problem (ESPP) asks us to find a minimum length path between two points on a 2D plane while avoiding a set of polygonal obstacles. Existing approaches for ESPP, based on Dijkstra or A* search, are primal methods that gradually build up longer and longer valid paths until they reach the target. In this paper we define an alternative algorithm for ESPP which can avoid this problem. Our approach starts from a path that ignores all obstacles, and generates longer and longer paths, each avoiding more obstacles, until eventually the search finds an optimal valid path.

Introduction
The Euclidean Shortest Path Problem (ESPP) is fundamental for applications such as computer video games (Algoor, Sunar, and Kolивand 2015). Here the operating environment (i.e., map) is often given as a set of obstacles and our task is to find a shortest path, from a start point \( s \) to a target point \( t \), all while avoiding intersecting any obstacles. One of the reasons ESPP is challenging to solve is that game worlds are often dynamic: between any two start-target queries, obstacles can be added, moved or removed.

Conventional ESPP methods include algorithms based on Visibility Graphs (VG) (Lozano-Pérez and Wesley 1979; Hong, Murray, and Wolf 2016) and Navigation Meshes (De- myen and Buro 2006; Cui, Harabor, and Grastien 2017), both of which must convert the Euclidean map to a discrete form more suitable for search. Creating and updating these discrete representations incurs additional costs which can dominate search time when the map changes frequently. More recent methods, such as RayScan (Hechenberger et al. 2020), do not incur any additional costs and they can offer competitive online performance. All of these methods can be described as primal solvers: that is, they all solve ESPP by growing valid optimal paths from \( s \) and exploring these paths in least-cost order, until one eventually reaches \( t \).

The main drawback of the primal approach is the inability to avoid fill in. This occurs when the search algorithm is forced to explore parts of the map that appear promising, but cannot possibly lead to an optimal solution. To mitigate fill-in, many works propose to improve the accuracy of the heuristic estimator; e.g., (Zhao, Taniar, and Harabor 2018; Shen et al. 2021). However these approaches usually depend on precomputed auxiliary data, which can be expensive to create and store, and which becomes entirely invalidated when the environment changes.

In this paper we describe Dual Pathfinding Search (DPS), a radically different approach to ESPP which searches as a dual problem. In dual search we always have a (super-)optimal path from start to target, but not one that is valid. We construct a tree of increasingly longer optimal but invalid paths until we find a valid path.

We then undertake an empirical study which shows that, in a range of settings, Dual Pathfinding Search can be substantially faster than currently leading ESPP methods, based on primal search. The paper is restricted to the case of convex polygonal obstacles (Rohnert 1986), see Figure 1.

Definitions
A path \( \pi \) is a string of vertices, from \( s \) to \( t \). Every adjacent pair forms an edge, which can be valid (intersects no obstacles), invalid (intersects obstacle(s)) or tentative (has not been checked). A path can also be valid (all edges valid), tentative (has a tentative edge) or invalid. A subpath \( \pi_{ab} \) is the substring of \( \pi \) between vertices \( a \) and \( b \).

We denote \( O_p \) as the obstacle incident to vertex \( p \). All vertices \( p \in \pi \) except \( s \) and \( t \) have an incident obstacle \( O_p \) that \( \pi \) bends around. Optimal paths are taut, that is they tightly bend around each incident obstacle without intersecting it. A direction \( D \) is either CW (clockwise) or CCW (counter-CW) in orientation, and \( \neg D \) swaps the orientation.

We define a \( D \)-curve as a subpath \( \pi_{ab} \) where with all substrings \( qwv \) of three vertices in \( \pi_{ab} \), vector \( q\vec{w} \) does not turn
Figure 2: Replacing edge $uv$ with either a path $CW$ around obstacle $O$ or $CCW$ around obstacle $O$.

$\neg D$ towards $uv$; i.e. the orientation is only going straight or turning in $D$-orientation. A subpath of only two vertices is considered both a $CW$- and $CCW$-curve.

Given an edge $uv$ and obstacle $O$, the $D$-bend $\mathcal{B}_{uv}(O)$ is defined as the shortest tentative $D$-curve from $u$ to $v$ around obstacle $O$ in direction $D$. Figure 2 has $\mathcal{B}_{uv}^{CW}(O) = ui jv$ (red dashed path) and $\mathcal{B}_{uv}^{CCW}(O) = umkv$ (blue dashed path).

**Dual Pathfinding Search**

DPS proceeds similarly to $A^*$ search, with two important differences: (i) DPS operates on tentative paths instead of vertices; (ii) DPS discovers obstacles by checking tentative paths for feasibility and then generates new successor paths by introducing detours around the discovered obstacles.

**A* Search:** DPS uses the tentative path length as an $f$-value. A shortest path is identified when a path popped of the queue is proven valid, which terminates the search.

**Expansion:** When a path $\pi$ is proven invalid, DPS produces two successor paths around the invalidating obstacle $O$, by essentially modifying $\pi$ to bend tightly around $O$ using a $D$-bend in each direction, then adjusting each path to maintain tautness.

Formally, we expand $\pi$ by first checking the validity of each edge, until one edge $uv$ is proven invalid by an obstacle $O$.

For each $D \in \{CW, CCW\}$, we do the following. Compute the maximal-length $D$-curve $C$ of $\pi$ that contains $uv$. The successor path $\pi^D$ will differ from $\pi$ only in that subpath $C$. Call $a$ and $b$ the first and last vertices of $C$. We first replace $uv$ in $\pi^D$ by the $D$-bend $\mathcal{B}_{uv}(O)$. We then modify $\pi^D$ by removing vertices between $a$ and $b$ (exclusively) so that the resulting subpath is taut (except possibly at $a$ and $b$).

We next consider the incident obstacle $O_a$ (if it exists), if corner $gyax \in \pi^D$ is not taut around $O_a$, then we have to bend $\pi^D$ around $O_a$ until it is taut. To do this, we replace $ax$ in $\pi^D$ with $\mathcal{B}_{ax}(O_a)$. If need be we do the same for $gybx \in \pi^D$ with $yb$. Path $\pi^D$ is now a successor.

Refer to Figure 2 for a successor example. Path $\pi = sq w v p t$ has edge $uv$ intersect $O$, successor $\pi^{CW}$ will first be assigned path $\pi$. Next we discover the maximal-length CW-curve including $uv$, giving us $\pi^{CW}_{qp} = qwvp$, the subpath that we will modify. We replace $uv$ with $\mathcal{B}_{uv}^{CW} = ui jv$ in $\pi^{CW}$ to give us $\pi^{CW} = squijvpt$, then correct for tautness between $q$ and $p$ by removing vertices $u$ and $v$ that violates it, giving $\pi^{CW} = sqijvp t$. Corner $qi$ is not taut around $O_q$, thus replace $qi$ with $\pi_{qi}^{CW}(O_q) = qzi$. Corner $jpt$ is taut around $O_p$, thus we do nothing. We now have the successor $\pi^{CW} = sqxijvp t$.

**Experiments and Conclusions**

We evaluate DPS on a set of 6 test maps (we show 3 in Figure 1). Our implementation is C++ (Hechenberger 2022) and compiled with g++ 11.1.0. Our test machine runs Arch-Linux (5.15.4), has 16 GB RAM (12 GB made available to algorithm) and an Intel Core i7-8750H CPU fixed at 2.2 GHz no turbo boost. For each map we solve 1000 instances (st-pairs). Test maps and implementations are available (Hechenberger 2022).

Results in Table 1 compare DPS against state-of-the-art methods Polyanya (Cui, Harabor, and Grastien 2017) and RayScan (Hechenberger et al. 2020). The $A^*$ expansion in DPS was done by choosing a random edge $uv$ and checking its validity until an invalid is determined. We then select the largest looking obstacle intersecting $uv$ by considering all obstructions $O$ and choosing the first vertex $q$ on $O$ in CW-bend and first vertex $r$ in CCW-bend on $O$, then taking the minimum of path length $sq uv$ and $ur v$, choosing the obstacle with the maximum of this value. DPS-R simply selects a random invalid edge then randomly chooses an obstacle blocking such edge.

As we can see, DPS can massively outperform its competitors is many instances, see particularly well with divide3 with a speedup of greater than 10. The selection of the obstacle, and to a lesser extent edge, is important for the runtime of DPS, as we see with random obstacle selecting in DPS-R, not beating DPS in anything and even getting out-of-memory on circle2. This is a result of DPS not being a polynomial algorithm in the worst case, though that requires unfavourable circumstances and is not common, as we see exponential runtime with these maps. Improvements on performance can be achieved with pruning rules, which we have not included in this extended abstract.

<table>
<thead>
<tr>
<th>maps</th>
<th>unit</th>
<th>DPS</th>
<th>DPS-R</th>
<th>Poly</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>ms</td>
<td>7.08</td>
<td>7.89</td>
<td>14.0</td>
<td>39.4</td>
</tr>
<tr>
<td>medium</td>
<td>ms</td>
<td>33.4</td>
<td>82.0</td>
<td>86.0</td>
<td>366</td>
</tr>
<tr>
<td>divide1</td>
<td>ms</td>
<td>32.1</td>
<td>109</td>
<td>444</td>
<td>668</td>
</tr>
<tr>
<td>divide3</td>
<td>ms</td>
<td>42.5</td>
<td>3480</td>
<td>1100</td>
<td>1860</td>
</tr>
<tr>
<td>circle1</td>
<td>ms</td>
<td>82.1</td>
<td>117</td>
<td>49.0</td>
<td>726</td>
</tr>
<tr>
<td>circle2</td>
<td>s</td>
<td>67.3</td>
<td>-</td>
<td>0.592</td>
<td>6.75</td>
</tr>
</tbody>
</table>

Table 1: Total runtime on each of our six test map. small is similar to medium with less objects, divide3 is similar to divide1 but with 3 dividers, and circle2 like circle1 with more (smaller) objects.
Acknowledgements
This work is partially supported by the Australian Research Council under grant DP200100025.

References


