Proceedings of the Twenty-Sixth International Conference on
Automated Planning and Scheduling (ICAPS 2016)

Strict Theta*: Shorter Motion Path Planning Using Taut Paths

Shunhao Oh and **Hon Wai Leong**
Department of Computer Science
National University of Singapore
ohoh@u.nus.edu, leonghw@comp.nus.edu.sg

Abstract

A common way to represent dynamic 2D open spaces in robotics and video games for any-angle path planning is through the use of a grid with blocked and unblocked cells. The Basic Theta* algorithm is an existing algorithm that produces near-optimal solutions with a running time close to A* on 8-directional grids. However, a disadvantage is that it often finds non-taut paths that make unnecessary turns. In this paper, we demonstrate that by restricting the search space of Theta* to taut paths, the algorithm will, in most cases, find much shorter paths than the original. We describe two novel variants of the Theta* algorithm, which are simple to implement and use, yet produce a remarkable improvement over Theta* in terms of path length, with a very small running time trade-off. Another side benefit is that almost all paths found will be taut, which makes more convincing paths.

Introduction

A popular way to represent dynamic 2D open spaces in robotics and video games for path planning is through the use of a uniform square grid (Algfoor, Sunar, and Kolivand 2015). An advantage of this representation is that the open space can be dynamic - obstacles can be added, removed or shifted easily by simply updating the boolean array to represent the new state of the grid. This is in contrast to other representations like the navigation mesh. Updating a navigation mesh in response to changes in the environment can be a complicated process.

The need for dynamic mazes is common in many applications. In Real-Time Strategy games, players are allowed to construct structures on the map. In robotics, a room's layout may change as objects like chairs are moved, changing the geometry of the map. These cases show the advantage of online pathfinding algorithms over offline algorithms. Offline algorithms are algorithms that make use of a costly preprocessing step, which may need to be repeated whenever the grid is updated, losing one of the main advantages of grid-based representations over navigation meshes.

Many fast online algorithms like the Jump Point Search Algorithm (Harabor and Grastien 2011) exist for computing optimal shortest paths when movement is restricted to the four cardinal directions and 45-degree diagonals. Any-angle paths however do not have this movement restriction, and can thus move in any direction across unblocked tiles. In addition to being shorter, any-angle paths also avoid having the unnecessary heading changes 8-directional paths have.

Optimal algorithms for any-angle pathfinding like A* on visibility graphs or Anya (Harabor and Grastien 2013) exist, but are much slower and more complex than 8-directional pathfinding algorithms. Online Heuristic algorithms like Theta* (Nash et al. 2007), however are simplier, run faster, in exchange for returning non-optimal paths. There are also offline heuristic algorithms like Block A* (Yap et al. 2011) and Subgoal Graphs (Uras and Koenig 2015b) which can, through the use of preprocessing, find short any-angle paths with faster running times than Theta*.

In this paper, we describe Strict Theta*, a variant of Theta*, that bears the same characteristics, while finding paths that are much closer to optimal than Theta*. An additional benefit is that almost all paths found by the algorithm are taut. While a taut path is not guaranteed to be optimal, taut paths are more convincing as a non-taut path is clearly suboptimal, due to the possibility of shortcutting a non-taut path to make a shorter path.

In practice, slight suboptimalities in the found path is often not an issue, but non-taut paths would contribute to the perceived irrationality of the agent, as the agent takes paths with clearly better alternatives. A good grid-based any-angle pathfinding algorithm is fast, can compute near-optimal paths, and is online. The Strict Theta* algorithm meets all of these requirements.

Basic Theta*

We first describe the Basic Theta* algorithm as our algorithm is an extension of Theta*. The principle behind Theta* is simple. The algorithm is based on the A* algorithm on 8-directional grids, with one small modification.

When relaxing a vertex u with successor v, two paths are considered, as shown in Figure 1. Path 1 is the path from vertex u to v, and Path 2 is the direct path from the parent p of u the vertex v. If the algorithm chooses Path 2 over Path 1, the parent of v will be set to the vertex p instead of the vertex u like A* in the relaxation step. By the triangle inequality, Path 2 is guaranteed to be no longer than Path 1. Path 2 is thus chosen if and only if there is line of sight from the vertex p to v.

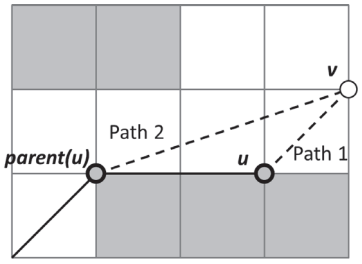

Figure 1: The two paths considered by Theta* when relaxing v from current vertex u.

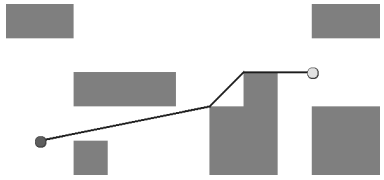

Figure 2: Theta* taking an suboptimal, non-taut path.

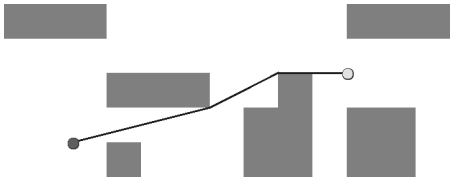

Figure 3: Taut paths are more believable, and are often shorter than non-taut paths.

Suboptimality of Basic Theta*

There are however many examples like Figure 2 that clearly show that Theta* is not an optimal algorithm. A common reason why these paths are not optimal they are not taut. A taut path is defined as a path where every heading change in the path "wraps" tightly around some obstacle.

The paths found by Theta* are often close to the optimal path length. However, the found paths may not be very convincing as there are clearly unneeded heading changes made in the paths. Some of these examples, like Figure 2 can be especially jarring, due to the large detour the agent makes to "touch" the opposite corner before making a sharp turn back. This is in contrast to the taut path in Figure 3. Though taut paths are not necessarily optimal, they are often shorter, and more believable.

Main Idea

Assuming the goal is reachable, there will be a taut path to the goal. Notably, the optimal path will always be taut. A simplified version of our algorithm is thus as follows: Use the Theta* algorithm, but before relaxing any vertex v with predecessor u, we first check if the sub-path $parent(u)$-u-v is taut. If it is not taut, we do not relax vertex v. The algorithm will thus be unable to construct any non-taut paths.

In terms of path length, this simple change improves performance significantly. However, as Theta* is not an optimal algorithm, and the algorithm restricts itself to finding only taut paths, this occasionally causes the algorithm to fail to find the goal.

To fix this, instead of not relaxing the vertex when the path from u to v is not taut, we do the relaxation as usual, but add a temporary penalty $b > 0$ to the path length. This penalty pushes the vertex v further back in the priority queue, and slightly longer taut paths towards v may also replace the found path to v. This penalty is removed when the vertex is visited. This algorithm, Strict Theta*, obtains the same results as before, but is always guaranteed to find the goal.

Strict Theta*

Strict Theta* is implemented the same way Theta* is implemented, except with an additional constant-time tautness check in the relaxation step. Before a vertex v is relaxed with parent u, the sub-path $(parent(u), u, v)$ is first checked for tautness. If the path is not taut, an additional penalty value is added to $distance(v)$ after relaxation. The vertex v is additionally marked as not taut, so that the increase in the g-value can be reversed later when the vertex v is extracted from the priority queue.

Algorithm 1 Update successor v from the current vertex u

```
1: procedure UpdateVertex(u, v)
2:     if LineOfSight(parent(u), v) then
3:         return Relax(parent(u), v)          ▷ Path 2
4:     else
5:         return Relax(u, v)                  ▷ Path 1
```

The pseudocode of Strict Theta* is separated into two parts. When a vertex u is removed from the OPEN list and explored, UpdateVertex is called on each of its successors v. Just like in Theta*, we check for line-of-sight from the parent of u to v. If there is line of sight, we call Relax on $parent(u), v$, corresponding to taking Path 2. Otherwise, we call Relax on u, v, corresponding to taking Path 1.

Algorithm 2 Attempt to update the parent of v to u

```
1: procedure Relax(u, v)
2:     newWeight ← distance(u) + c(u, v)
3:     if newWeight < distance(v) then
4:         if IsTaut(parent(u), u, v) then
5:             distance(v) ← newWeight
6:             parent(v) ← u
7:             mark v as taut
8:         else
9:             distance(v) ← newWeight + PENALTY
10:            parent(v) ← u
11:            mark v as not taut
12:        return True
13:    return False
```

Checking whether a sub-path (w, u, v) is taut only requires a single tile to be checked. From the four tiles adjacent to the vertex u, we check the tile on the side containing the non-reflex angle $\angle wuv$. The sub-path is taut if and only

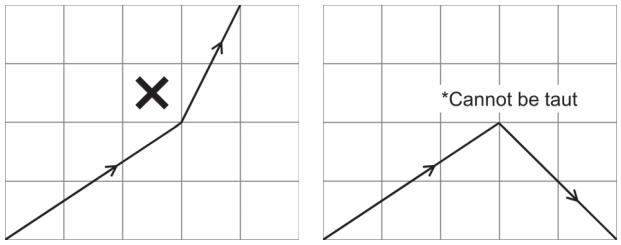

Figure 4: A single obstacle is checked to determine tautness.

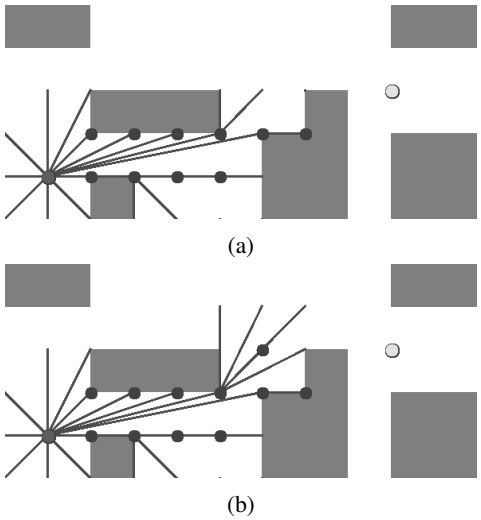

Figure 5: Decision point where Strict Theta* chooses the taut path, over the longer, non-taut path.

if this tile is blocked (Figure 4a). If $\angle wuv$ is acute, the path cannot be taut (Figure 4b). If (w, u, v) is a straight line, the path is always taut.

This simple modification corrects many of the errors made by Theta*, like the path in Figure 2. As shown in Figure 5, due to Strict Theta*'s preference for the taut path at this decision point, it is able to find the optimal path (Figure 3) to the goal. Note that while the algorithm is not guaranteed to find a taut path, it has a low probability of finding a non-taut path towards the goal.

Recursive Strict Theta*

The algorithm can be further improved by using a recursive form of Strict Theta*. When relaxing a vertex v with parent u, instead of considering only the two paths (u, v) and $(parent(u), v)$, we search backward from u to find the first ancestor w (with line-of-sight) where $parent(w)$-w-u is taut. Notably, if $(parent(u), u, v)$ is taut from the start, Path 2 from the algorithm will not even be considered.

If we lose line-of-sight before we find a taut parent, we assign the last node with line-of-sight as the parent of u, add a penalty value to the distance of v, and mark v as not taut.

A side effect of stopping the search when a taut path is found is that because collinear points also make a taut path,

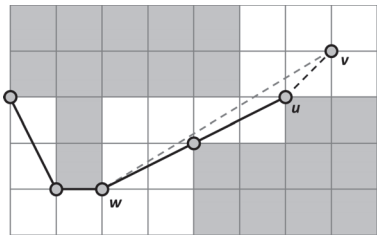

Figure 6: In this example, the parent of v would be set to w by the Recursive Strict Theta* algorithm.

Algorithm 3 Update successor v from the current vertex u

```
1:  procedure UpdateVertex(u, v)
2:      if IsTaut(parent(u), u, v) then
3:          return Relax(u, v, True)
4:      else
5:          if LineOfSight(parent(u), v) then
6:              return UpdateVertex(parent(u), v)
7:          else
8:              return Relax(u, v, False)
```

Algorithm 4 Attempt to update the parent of v to u

```
1:  procedure Relax(u, v, isTaut)
2:      newWeight ← distance(u) + c(u, v)
3:      if newWeight < distance(v) then
4:          if isTaut then
5:              distance(v) := newWeight
6:              parent(v) := u
7:              mark v as taut
8:          else
9:              distance(v) ← newWeight + PENALTY
10:             parent(v) ← u
11:             mark v as not taut
12:         // collinear path optimisation
13:         if IsCollinear(parent(u), u, v) and
14:             ¬IsOuterCorner(u) then
15:             parent(v) ← parent(u)
16:         return True
17:     return False
```

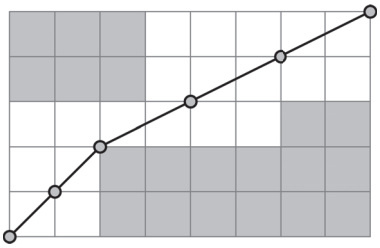

Figure 7: Taut Path with multiple collinear points.

a single straight-line path could be split into many small segments of collinear points (Figure 7). This makes the algorithm run slower due to an increased recursion depth, meaning more line-of-sight checks.

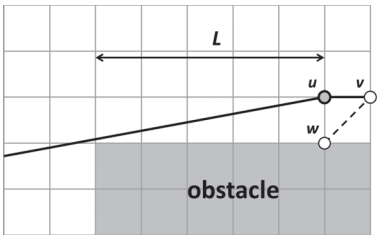

Figure 8: Strict Theta*, with a sufficiently high penalty value, finds a shorter, taut path to v through exploring w.

To remedy this, in the Relax step, when setting the parent of v to u, if the path $parent(u)$-u-v is collinear, $parent(u)$ is connected directly to v instead. This inductively removes all collinear points from the path found. Note that collinearity checks are constant-time operations. However, we make an exception for when a collinear point u lies on an outer corner of an obstacle. Points at outer corners are important to keep as they allow the path to pivot around obstacles.

Choosing the Penalty Value

The results of the algorithm vary based on the penalty value. A penalty value close to 0 makes the algorithm behave exactly like Theta*, and a penalty value too high can occasionally make the algorithm prefer to construct a much longer path over a shorter, non-taut path to the goal. Assuming the straight-line distance heuristic is used, penalty values above $\sqrt{2}-1$ (around 0.42) make a significant improvement in the algorithm's performance, and increasing the penalty value past $\sqrt{2}-1$ yields little further benefit.

This can be explained by the most common way Strict Theta* finds alternate, shorter taut paths than Theta*. Referring to Figure 8, the algorithm finds a non-taut path to the vertex v via u, and thus adds a penalty to v. The parent of vertex w is on the outer corner on the top left of the obstacle. If the algorithm holds off on exploring vertex v until w is explored, a taut path from the parent of w to v will be found. However, as vertex w is out of the way, Theta* is unlikely to explore w before vertex v. Using the straight-line-distance heuristic, the heuristic weight $h(w)$ of w is at most $\sqrt{2}$ more than that of v, and thus the f-value of w is approximately $(L+\sqrt{2})-(\sqrt{L^2+1}+1)$ more than the f-value of v, which is an increasing sequence with limit $\sqrt{2}-1$. A penalty value of $\sqrt{2}-1 \approx 0.42$ is thus sufficient for vertex w to be explored before vertex v.

Results

The algorithms are compared over both randomly generated grids and game maps. The penalty values of both versions of Strict Theta* are fixed to $0.42 \approx \sqrt{2}-1$. The results in terms of path length, percentage taut, percentage optimal, and running time are given in Tables 1 to 4.

The algorithms tested are Basic Theta*, Strict Theta* (S.Theta*), Recursive Strict Theta* (RS.Theta*). Theta* with post-smoothing (Botea, Müller, and Schaeffer 2004) is also included for comparison. Post-smoothing is a post-processing step that improves path lengths and the percentage of taut/optimal paths using negligible extra time.

	Theta*	Theta*PS	S.Theta*	RS.Theta*
Rand 6%	1.000725	1.000708	1.000431	1.000262
Rand 20%	1.002020	1.001911	1.000320	1.000137
Rand 40%	1.001685	1.001506	1.000140	1.000077
Obstacles	1.002269	1.002112	1.001339	1.000708
Maze	1.000379	1.000379	1	1
Game	1.000506	1.000466	1.000014	1.000003

Table 1: Average path length as a ratio to the optimal.

	Theta*	Theta*PS	S.Theta*	RS.Theta*
Rand 6%	2.65	2.65	3.11	3.33
Rand 20%	5.89	5.90	6.23	6.65
Rand 40%	17.84	17.91	18.98	20.52
Obstacles	9.08	9.04	9.65	10.12
Mazes	71.48	71.18	74.24	79.51
Game	11.08	11.01	11.64	12.21

Table 2: Average running time (in ms) averaged over 450 runs per test case.

	Theta*	Theta*PS	S.Theta*	RS.Theta*
Rand 6%	0.173	0.220	0.867	0.993
Rand 20%	0.033	0.067	0.607	1
Rand 40%	0.067	0.113	0.573	1
Obstacles	0.050	0.100	0.650	1
Maze	0.000	0.000	1	1
Game	0.581	0.733	0.939	1

Table 3: Percentage of paths found that are taut.

Algorithms like A* and Field D* have not been included as Theta* has been shown to give superior path lengths to these algorithms (Uras and Koenig 2015a). A Recursive version of Theta* was also tested, but it consistently obtained poorer results than even Theta*, in terms of both path length and running time. The pathfinding algorithms were implemented in Java on a 2.60 GHz Intel i5 processor.[1]

Maps

The maps used are divided into categories. Randomly generated maps used are of size 500x500, with either 6%, 20% or 40% blocked tiles. "Obstacles" maps and maze maps are of size 512x512. Game maps are taken from games like Baldur's Gate, Starcraft and Warcraft III.[2]. Planning is simpler on game maps, due to their reduced complexity.

Each test case is a randomly selected pair of start and end points one of the maps tested on. There are a total of 1000 test cases: 150 each for Rand 6%, Rand 20% and Rand 40%, 60 for Obstacles, 100 for Mazes, and 390 for Game Maps.

[1]The implementations are available at github.com/Ohohcakester/Any-Angle-Pathfinding

[2]"Obstacles", maze and game maps are taken from Nathan Sturtevant's Moving AI Lab benchmarks (Sturtevant 2012) at http://www.movingai.com/benchmarks/

	Theta*	Theta* PS	S.Theta*	RS.Theta*
Rand 6%	0.067	0.073	0.280	0.440
Rand 20%	0.027	0.053	0.313	0.587
Rand 40%	0.067	0.113	0.407	0.713
Obstacles	0.000	0.000	0.017	0.117
Maze	0.000	0.000	1	1
Game	0.584	0.733	0.934	0.992

Table 4: Percentage of paths found that are optimal.

Path Quality

All path lengths found are computed as a ratio to the optimal path length, computed using A* on Visibility Graphs, a known optimal algorithm (Lozano-Pérez and Wesley 1979). As shown in Table 1, the path lengths found by Strict Theta* are a lot closer to optimal than those found by Theta*. Recursive Strict Theta* improves this even further.

Strict Theta* also produces a much higher proportion of taut and optimal paths than Theta*. Furthermore, out of the 1000 test cases, only one path found by the Recursive Strict Theta* algorithm was not taut.

We see the greatest improvement in dense maps. Notably, Theta* performs better on sparse maps than on dense maps, while Strict Theta* performs better on dense maps than on sparse maps. This is as restricting the search space to taut paths leaves few suboptimal paths in dense grids. Interestingly, from the 100 test cases over the 5 maze maps, almost none of the paths found by the Theta* are optimal, while all of the paths found by both Strict Theta* are optimal.

Post-smoothing is able to slightly improve path lengths and percentage taut for both Theta* and Strict Theta* that are taut, the improvement is however very small, and nowhere near that of Strict Theta* over Theta*.

Running Time

We can see from Table 2 that there is a slight cost to runtime when the Strict Theta* algorithms are used over Theta*. Strict Theta* runs slightly slower than Theta*, and Recursive Strict Theta* runs slightly slower than Strict Theta*.

Empirically, the running times of Strict Theta* as a ratio to Theta* is between 95% and 108%. The running times of Recursive Strict Theta* as a ratio to Theta* is between 103% and 116%. While Theta* and Strict Theta* execute a constant number of line-of-sight checks per relaxation, Recursive Strict Theta* does not. However, the slowdown appears to be by a constant factor, and is a small trade-off compared to improvement in path quality over Theta*.

Conclusions

We can see that Strict Theta*, a simple change from Theta* and thus does not take significant time to implement, is able to find significantly shorter paths than Theta*, with a small cost to running time. The Recursive Strict Theta* algorithm improves this even further, and has the added bonus of almost always finding a taut, and thus believably optimal path towards the goal.

Anya, an optimal any-angle pathfinding algorithm (Harabor and Grastien 2013), restricts its search space to taut paths as we know that the optimal any-angle path is guaranteed to be taut. We have shown that, even in non-optimal algorithms, the idea of taut path restriction can be applied to construct much shorter paths, at little runtime cost. A possible direction of further work would be to try applying similar ideas to other online or offline heuristic algorithms to find shorter and more taut paths with litle runtime cost.

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*.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1(1):1–22.

Harabor, D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *AAAI*.

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.

Lozano-Pérez, T., and Wesley, M. A. 1979. An algorithm for planning collison-free paths among polyhedral obstacles. *Communications of the ACM* 22:560–570.

Nash, A.; Daniel, K.; Koenig, S.; and Felner, A. 2007. Any-angle path planning on grids. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1117–1183.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.

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

Uras, T., and Koenig, S. 2015b. Speeding-up any-angle path-planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Yap, P.; Burch, N.; Holte, R.; and Schaeffer, J. 2011. Block a*: Database-driven search with applications in any-angle path-planning. In *AAAI*.