University of Huddersfield Repository

Gregory, Peter and Chrpa, Lukáš

Abstract Augmented Lazy Theta*: Applying Graph Abstractions to Trajectory Planning with Speed-Limit Constraints

Original Citation


This version is available at http://eprints.hud.ac.uk/20863/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/
Abstract Augmented Lazy Theta*: Applying Graph Abstractions to Trajectory Planning with Speed-Limit Constraints

Peter Gregory  
Digital Futures Institute  
School of Computing  
Teesside University, UK  
p.gregory@tees.ac.uk

Lukáš Chrpa  
PARK research group  
School of Computing and Engineering  
University of Huddersfield, UK  
l.chrpa@hud.ac.uk

Abstract

Trajectory planning is a generalisation of path planning in which velocity is also taken into consideration. Considering velocity, in the context of a real-world system such as a robotic vehicle, means considering physical constraints such as the speed limit constraints when steering the vehicle. One of the recent algorithms for solving this type of planning problem is Augmented Lazy Theta*. In this work we introduce the trajectory planner Abstract Augmented Lazy Theta*. This planner uses a combination of abstraction and relaxation to reduce the search space of Augmented Lazy Theta*. We demonstrate that Abstract Augmented Lazy Theta* significantly improves the time performance of Augmented Lazy Theta* whilst retaining the same overall solution quality. Additionally, we show that Abstract Augmented Lazy Theta* performs competitively with another state-of-the-art trajectory planning algorithm, the RRT* algorithm, by demonstrating significantly better overall performance, in terms of both time and quality.

1 Introduction

Efficient path planning for robotic, and simulated, vehicles requires reasoning about the velocity of the vehicle. It is sensible to drive as quickly as possible in order to reach the destination quickly. However, there are constraints on the speed a vehicle may travel: specifically the vehicle’s angle of turn is limited by its speed (lower speeds are necessary for tighter turns). We call this more general problem (where velocity is also taken into consideration) and adding new constraints to the problem naturally makes trajectory planning more difficult than traditional path planning. In this work we present a trajectory planning algorithm, called Abstract Augmented Lazy Theta* (AALT* from here), which uses a combination of abstraction and relaxation in order to increase the efficiency of a current approach to this problem (Augmented Lazy Theta*, also referred to as ALT*).

We compare the performance of AALT* with both the ALT* and RRT* (RRT* or Rapidly-growing Random Trees*: a state of the art trajectory planning approach). We test the planners on 240 tasks across six different maps of varying structure. Our results show that the AALT* algorithm is significantly faster than ALT*, whilst finding plans in comparable quality (supporting the idea that abstraction is an effective approach for trajectory planning). We also demonstrate that, although RRT* finds its first solution faster than AALT* on some maps, AALT* finds plans of significantly higher quality than RRT* even when RRT* is allowed more time.

2 Problem Definition

Trajectory planning deals with a problem of finding a trajectory from a configuration of origin to a configuration of destination following constraints such as obstacle avoidance or maximum speed for turn manoeuvres. In the robotic literature, configurations are defined by position and velocity vectors. Path planning, which is widely used in video games, relaxes the dynamic part of trajectory planning, that is, planning for one position to another while avoiding obstacles. Continuous (Euclidean) space is often discretized into (regular) grids, where grid cells are either free or blocked.

Formally, a grid can be represented by an undirected graph $G = (V,E)$, where vertices $V$ represent free grid cells and edges $E$ connect neighbouring (free) cells. A path $p = (v_0, \ldots, v_k)$ is a sequence of vertices ($v_i \in V$) such that $(v_i, v_{i+1}) \in E$ ($0 \leq i < k$). Note that the constraint for adjacent vertices in a path can be weakened by introducing a line-of-sight (LOS) check which, informally said, refers to situations when more distant vertices are ‘visible’. More formally, $LOS(v_i, v_j)$ is true if and only if the line connecting $v_i$ and $v_j$ does not cross any blocked cell (or obstacle). LOS hence allows any-angle path planning and provides more realistic paths. A path planning problem is a triple $P = (G,v_0,v_g)$ where $G = (V,E)$ is an undirected graph representing the underlying grid, $v_0 \in V$ is the initial position (cell) and $v_g \in V$ is the goal position (cell). A solution of a path planning problem $P = (G,v_0,v_g)$ is a path $p = (v_0, \ldots, v_g)$.

For trajectory planning we have to provide an extension of the terminology. A configuration $c$ is specified by position $pos(c)$ and velocity vector $vel(c)$ (velocity vector incorporates orientation and speed). A trajectory is a continuous function of time to the (Euclidean) space. Given the discretization of the space into grids, we split trajectories into segments, concretely Straight manoeuvre (a line) and Turn manoeuvre (an arc). The line-of-sight (LOS) check between configurations is the same as in the path planning case be-
cause only positions are needed. The LOS check is insufficient for checking whether we can transit from one configuration to another, hence we introduce speed limit (SL) check which takes into the account also velocities. In other words, the SL check verifies whether, for instance, the entity is moving slowly enough to perform a Turn manoeuvre, or can accelerate to reach the target speed. The notion path is extended in such a way that it is a sequence of configurations \((p = (c_0, \ldots, c_k))\) such that \(LOS(c_i, c_{i+1})\) and \(SL(c_i, c_{i+1})\) holds for \(0 \leq i < k\). A trajectory planning problem is a triple \(P = (C, c_0, c_g)\) where \(C\) is a set of configurations, \(c_0 \in C\) is an initial configuration and \(c_g \in C\) is a goal configuration. A solution of a trajectory planning problem \(P = (C, c_0, c_g)\) is a path \(p = (c_0, \ldots, c_g)\).

Given a solution path of a trajectory planning problem, the trajectory can be constructed from Straight and Turn manoeuvres in the following way. Turn manoeuvres are kept within the cells because then we can easily be sure that the Turn manoeuvre will not cross any obstacle. Let \(r_i\) be a radius of cell inscribed circle and \(\alpha\) be a turn angle. Then the radius of the Turn manoeuvre \(r\) is calculated as: \(r = r_i \cot \frac{\alpha}{2}\). Straight manoeuvres are used to transit between successive configurations \(c_i\) and \(c_{i+1}\) in the solution path. They are not thus connecting the exact positions of \(c_i\) and \(c_{i+1}\) referring to centers of cells (except initial and goal cells) but they start at \(r_i\) distance from \(pos(c_i)\) and finish at \(r_i\) distance of \(pos(c_{i+1})\). Turn manoeuvres thus provide a smooth connection of the adjacent Straight manoeuvres. Speed can be adjusted only for Straight manoeuvres which is done by applying uniformly accelerated motion.

The configuration space can be constructed in a similar way as the graph representing a grid, i.e., the vertices are configurations and edges connect configurations in neighbouring (free) cells and have ‘compatible’ velocity vectors. However, the number of configurations might increase rapidly when any-angle path planning methods (e.g. Theta* (Daniel et al. 2010)) are applied. A similar problem might occur when a range of possible speed values is large or densely discretized. Therefore, for practical reasons the underlying graph for the search will remain the same as for path planning (i.e. positions only) and velocity vectors will be calculated during the search. This is discussed in detail later in the text.

3 Background

This work extends ideas in path planning with graph abstractions to trajectory planning with any-angle planning and speed limit constraints. Before discussing our algorithm, we provide a background to these areas.

3.1 Graph Abstractions

Abstraction is a technique widely used to simplify problem solving, in such areas as Planning, Constraint Programming, Databases and Machine Learning (Saitta and Zucker 2013). It is the process of removing information in order to reduce the size or complexity of a problem, whilst retaining enough structure to allow the results of reasoning about the abstract problem to still have some meaning in the original problem. Abstraction has been successfully applied in grid-based path planning (Botea, Müller, and Schaeffer 2004; Sturtevant and Jansen 2007). Bäckström and Jonsson (2013) identify a recent trend towards refinement techniques in domain independent planning (Gregory et al. 2011; Bäckström and Jonsson 2013; Seipp and Helmert 2013). It has been shown to drastically reduce the search time required to find solutions, whilst only reducing solution quality by a small amount.

The typical way in which abstraction is used in path planning is by abstracting the underlying graph and then using the abstraction to limit the search space in a two-step procedure. In the first of these steps a hierarchy of increasingly abstract graphs is created. An example of an abstraction hierarchy is given in Figure 1, where the ground problem (we use the term ground to refer to the actual problem under consideration) is in the top left and the most abstract is at the bottom right. The second step is to solve these path planning problems in reverse order, where the solution to each abstract problem is used in order to reduce the size of the search space at the next-most ground level by considering only those nodes that are subsumed by the nodes in the abstract solution.

3.2 Theta* and Lazy Theta*

A* is probably the best known algorithm for informed search. In path planning A* can be easily applied if the (Euclidean) space is discretized and represented by a graph. A* maintains a priority queue \(open\) where the elements are ordered according to their value (the element with the smallest value is on the front) which is calculated as a sum of \(g(v)\), the actual cost from the initial position to \(v\), and \(h(v)\), the estimated cost from \(v\) to the goal position. It starts by putting the initial position into \(open\). In an intermediate step a loop is performed until \(open\) is empty (there is no solution): \(v\) is taken from the front of \(open\), if \(v\) is the goal position then the solution is returned, otherwise \(v\) is put into a \(close\) list and then \(v\) is expanded. The expansion of \(v\) is done in such a way that for each of its neighbours \(v'\) we check whether \(v'\) is in \(close\), if not \(g(v')\) and \(h(v')\) are calculated and \(v'\) is put into \(open\) or its value is updated if smaller. We also keep a track of predecessors, so we set \(parent(v') = v\) (if \(v'\) is added into or updated in \(open\)).

Solutions produced by A* are, however, not very realistic since we can transit only between neighbouring cells in grids. Such solutions can be improved in a post-processing step by iterative checking whether the LOS check is satisfied between the given cell and its grand-parent (i.e. \(parent(parent(v))\)) (Botea, Müller, and Schaeffer 2004). This idea is exploited in the Theta* algorithm (Daniel et al. 2010) where rather than improving solutions in a post-processing step the LOS check is performed during the search. Theta* is a variant of A* but when \(v\) is being expanded for all its neighbours \(v'\) we perform the LOS check with \(parent(v)\), i.e., \(LOS(parent(v), v')\). If the LOS check succeeds, then \(parent(v') = parent(v)\), otherwise Theta* behaves in the same way as A*. Although Theta* generates solutions which are realistic, optimality (in the continuous space) is not guaranteed (Daniel et al. 2010).

Lazy Theta* (Nash, Koenig, and Tovey 2010) is a vari-
Figure 1: An example graph abstraction hierarchy on a grid-structured map. Nodes are shown as continuous regions of ground cells with the same shade. Each abstract node subsumes several nodes from the previous level in the hierarchy. The ground problem (top left) has approximately 1000 nodes whilst the most abstract shown (bottom right) has approximately 10 nodes.

ant of Theta* which performs lazy evaluation of the LOS checks. In other words, because the LOS check might be computationally expensive it is performed only when necessary. In the expansion phase, it is, therefore, assumed that the LOS check is successful. Of course, such an assumption may be incorrect. This is verified when \( v \) is taken from the front of open. If \( \text{LOS}(\text{parent}(v),v) \) fails, then it is necessary to revert the situation to the A* case which is done by selecting \( v' \) from close such that \( v \) is neighbour of \( v' \) and \( g(v') + h(v') \) is minimum. Then we set \( \text{parent}(v') = \text{parent}(v) \) and \( \text{parent}(v) = v' \). Clearly, this approach might lead to non-optimal solutions, especially when such reverts are frequent. This issue can be handled by re-inserting \( v \) into open if the LOS check fails which is incorporated in Lazy Theta*-R (Nash, Koenig, and Tovey 2010). However, empirical study (Nash, Koenig, and Tovey 2010) shows that using Lazy Theta*-R improves the quality of solution path only marginally while runtime increases more significantly (in comparison to Lazy Theta*).

3.3 Speed Limit Constraints Planner

The idea of extending path planning methods for trajectory planning has been thoroughly discussed by Chrpa and Osborne (2013). Incorporating velocities directly in search space may cause its explosion. Therefore, the advantage of reasonably small search space we deal with in path planning where only positions (grid cells) are considered should be kept. In other words, search space should be of similar size for both path and trajectory planning. This is provided by computing velocities in the expansion phase of A* (Theta* and Lazy Theta*), which is discussed later.

Velocity vectors are composed from orientations and speeds. Orientation is computed as a normalised difference between positions of successive configurations in a path, i.e., \( ||\text{pos}(c_{i+1}) - \text{pos}(c_i)|| \). Speeds are maintained as intervals rather than single values since it might lead to significant growth of the search space. In other words, introducing speed intervals allows us to process a set of configurations (having the same position and orientation) in a single step. For example, a car initially goes 30mph. Then, it continues to go straight ahead and at some point it can go between 10mph and 55mph (depending on whether the car is accelerating or decelerating). Then, the car may continue to go straight ahead (there is no limitation on speed in which this manoeuvre can be performed), or perform a Turn manoeuvre which can be performed in a maximum speed of 20mph. If the Turn manoeuvre is selected, then the current speed interval is trimmed, so the car can go between 10 and 20mph. Note that if for some very sharp Turn manoeuvre the maximum speed is 5mph, then it cannot be performed.

Speaking more formally, path planning methods are extended as follows. For A* and Theta*, velocities (orientations and speed intervals) are calculated in the node expansion phase. If \( c_i \) is being expanded and \( c_j \) is its neighbour (i.e., the position of \( c_j \) is in adjacent cell to \( c_i \)), then Turn and Straight manoeuvre between \( c_i \) and \( c_j \) are computed and then the SL check is performed. The SL check consists of determining interval of speed values (the speed limit constraint) for which it is possible to transit from \( c_i \) (resp. \( \text{parent}(c_i) \)) to \( c_j \). If the interval is disjoint with the current speed interval in \( c_i \) (resp. \( \text{parent}(c_i) \)) the SL check fails (\( c_j \) will not be
inserted to \textit{open} at this stage). Otherwise, the current speed interval in $c_j$ is calculated and $c_j$ is inserted into \textit{open}. Note that the current speed interval in $c_j$ is calculated from the speed interval which is an intersection of the current speed interval in $c_i$ (resp. $\text{parent}(c_j)$) and the speed limit constraint. For Theta*, if either of the LOS or SL checks fails, the ‘A* branch’ is performed (it also includes the SL check). For Lazy Theta*, it is in the node expansion phase assumed that the SL check is successful as well as the LOS check. Since this assumption might be also incorrect, the SL check is performed after the node (configuration) is selected for expansion. If the SL check fails, then the situation is reverted to the ‘A* case’. The SL check, however, must be performed also for the ‘A* case’. If it fails again, then the node (configuration) cannot be processed at this stage and the next node in \textit{open} is selected for expansion.

Extension of A* and (Lazy) Theta* as presented here is referred in literature to lite versions (Chrpa and Osborne 2013). Of course, the presented approach is generally incomplete, since after processing a configuration, we cannot revisit the same position (cell) even with a different velocity vector. On the other hand, it has been empirically shown that losing solvability of the problem occurs rarely and that such an approach performs similarly well as the path planning methods.

3.4 Related Work

Trajectory planning with vehicle dynamics is an important problem in the robotics community and is, hence, well-studied. In robotics, regular lattice discretization is often used since it better corresponds with manoeuvrability of robots, hence trajectories produced by these approaches can be very precise (Pivtoraiko, Knepper, and Kelly 2009). Trajectory planning had an important role in the DARPA Urban Challenge (Ferguson, Howard, and Likhachev 2009) by utilising an approach based on numerical optimisation which considers also effects of terrain and robot dynamics (Howard and Kelly 2007). SIPP (Phillips and Likhachev 2011) is a trajectory planning approach for environments with moving obstacles. SIPP introduces safe intervals, time periods for configurations without any collisions. Using safe intervals is a similar idea to using speed intervals resulting in ‘saving’ one dimension. There are exact and approximate algorithms, depending on whether optimal solutions are required. Probabilistic Roadmaps (Kavraki et al. 1996) and Rapidly-growing Random Trees (LaValle and Kuffner 1999) are two examples of incomplete search algorithms based on sampling to provide discretisations of continuous search spaces.

\textbf{RRT*} We focus in more detail on one other state-of-the-art trajectory planning approach, which we will use in order to compare with our algorithm. This is the \textbf{RRT*} algorithm.

\textbf{RRT*} (Karaman et al. 2011) is an extension to the RRT (LaValle and Kuffner 1999) algorithm and represents another state-of-the-art trajectory planner to compare against. RRT (or Rapidly-growing Random Trees) is a path-planning algorithm that quickly explores a state-space by building a tree from an initial location by repeatedly sampling arbitrary states and connecting them to the existing tree, through the closest location, when possible. \textbf{RRT*} improves on this algorithm by firstly connecting the sampled states to the state which minimises the distance travelled from the root of the tree, and secondly by reconnecting existing nodes in the tree through the sampled state if that reduces the distance travelled from the root of the tree.

4 Abstract Augmented Lazy Theta*

We now introduce the Abstract Augmented Lazy Theta* (AALT*) algorithm which uses the techniques of relaxation and abstraction in order to improve the ALT* algorithm with speed limit constraints described in section 3.3. The pseudo-code of the algorithm is provided in algorithm 1. The algorithm has three stages:

1. Relax the problem to a path planning problem (i.e. only positions are considered) and solve the relaxation using an abstraction-based path planning algorithm. Retain the ground states contained in each of the abstract solutions. Call the states visited in the concrete relaxed plan the relaxed state set.

2. Using the ALT* trajectory planning algorithm, solve the full problem, whilst only expanding states corresponding to locations within the relaxed state set.

3. If no plan is found, add the ground states from the next-most abstract plan to the relaxed state set, and then repeat step 2. Continue in this way until the problem is solved.

The solutions to the abstract problems in this case are solutions to a relaxed version of the problem with no velocities, and hence no speed limit constraints. The abstraction is therefore, in the language of Giunchiglia and Walsh (1992), a \textit{theorem increasing} abstraction (i.e. the abstraction permits solutions that are not possible in the concrete problem).

The states visited in the relaxed plan then form the locations that can be visited by the ALT* phase of the algorithm. Since the abstraction is theorem increasing, it is possible that there is no solution to the ALT* problem using only the cells visited in the abstract plan. Failure of this
kind can happen, for example, if the abstract plan requires turning a tight corner not permitted by the speed-limit constraints. In this case, a greater number of cells are considered by incorporating the cells visited in the next-most abstract solution. In the worst case, the cells in the most abstract solution (i.e. all of the cells in the map) will eventually be incorporated. Hence, although incomplete, full ALT* will eventually be executed and so its practical runtime performance is preserved.

Combining Abstraction with Line-of-Sight

One interesting consequence of combining a Theta*-based trajectory planner and an abstraction approach is that although the expansion phase of ALT* can only consider nodes that were part of the abstract solution, the line-of-sight check is still through the original ground space. Because of this, it is possible to find shortcuts that travel through free cells not in the abstract plan, so long as they start and finish within cells in the abstract plan. This is seen in the bottom pane of figure 2 where the ground plan takes a shortcut through space unoccupied by the relaxed solution. It is better for the ground solution to use fewer turn maneuvers, since these are the operations that require slowing the vehicle. Following the relaxed plan exactly would require far more turn maneuvers than the plan found by AALT* and so would have a longer duration.

5 Experimental Setup

In order to demonstrate the effectiveness of AALT* in terms of its time efficiency and quality, we provide an empirical analysis comparing it with both the ALT* algorithm and the RRT* algorithm. Since AALT* is based upon the ALT* algorithm, it is important to demonstrate that it improves upon it. It is also important to show that AALT* performs competitively with current state-of-the-art algorithms (where RRT* is taken here as an example of the current state of the art). We now outline our methodology for testing our hypotheses.

5.1 Benchmarks

In order to test the implementation, we use the benchmarks supplied by the Moving AI Lab at the University of Denver (Sturtevant 2012). This collection of instances are grid-based path planning problems on large maps derived from computer game maps and artificially constructed maze-like maps. We selected six maps from the benchmark suite that have varied layouts: these are shown in Figure 3. The maps are specified as $n \times m$ grids, where each cell can either be traversable or blocked. In Figure 3 the traversable cells are white and the blocked locations are black. We translate these maps to trajectory planning benchmarks by interpreting each cell as a $10 \times 10$ square, providing a $10n \times 10m$ map (i.e. the map is simply scaled up by 10 times).

5.2 Experimental Design

We compare the algorithms on 40 randomly generated instances for each benchmark map. We report the numbers of unsolved instances for each map and overall. In practice, all unsolved instances are by the RRT* algorithm; as a local search algorithm, it sometimes fails to find a solution within the termination criteria. Any instances found unsatisfiable in the relaxation of AALT* (guaranteed to be unsolvable

Figure 2: A presentation of the different stages in the AALT* algorithm. An abstract solution is shown in the top map (the abstract states which form the plan are displayed in darker colours and are roughly the states around the outside edge of the map). The ground relaxed solution is shown (middle) and the ground solution is shown in the bottom map, as the dashed line.
as this tests reachability on the path planning version of the problem) are ignored as RRT* is a local-search algorithm and can never report that the instance is unsatisfiable.

Once results were gathered, we tested the normality of the distributions of the differences in the data using the Shapiro-Wilk test for normality. Neither time taken to solve nor solution quality is distributed normally in either the raw data or the log of the data. Therefore in order to compare quality of plan and time taken to solve, we use the non-parametric Related-Samples Wilcoxon Signed Rank Test that tests for significant differences in the distributions of ranks with no requirement of normality.

5.3 Implementation, Hardware and Limits
All of the algorithms are implemented in a Java-based simulation system, and are run using the OpenJDK 1.7.09 virtual machine. The experiments are performed on a desktop PC with two Intel Xeon E5-2665 8-core CPUs with 64GB system memory, running CentOS Linux 2.6.32. Statistics are calculated using SPSS v21 Mac OSX. We impose no time limits on the algorithms. We place a 10,000 node limit on RRT*: although any runs that reach this limit are implicitly recorded in the main results, we nevertheless report the failure rate of RRT* to provide the reader with an idea of which maps were challenging for that algorithm. The objective function is the duration of the plan (we consider this as the most natural objective function; since velocities form part of the state, the planners should find the fastest plans rather than simply the shortest distance).

6 Experimental Evaluation
In this section, we report the findings of our experiments. Table 1 provides a summary of the results of the statistical testing for significance at the $p < 0.05$ level. The table shows the effect size ($r$ in the table), where $r = 0.1$ indicates small effect, $r = 0.3$ indicates medium effect and $r \geq 0.5$ indicates large effect. In the table, where $r$ is positive, performance has improved by using AALT*. When $r$ is negative, performance is degraded by using AALT*. Finally we indicate whether the hypotheses were supported for each map and also when all results are combined.

(H1) AALT* finds solutions faster than ALT*
H1 is supported across all maps and for the combined result. AALT* is significantly faster than ALT* on the combined result ($Z = 13.37, p < 0.001, r = 0.86$) with a large effect size. In fact, in almost every instance, AALT* finds solutions faster than ALT*.

(H2) AALT* finds solutions at least the same qual-
### Table 1: Summary of the experiments.

Computed data about the distributions of both runtime and quality are shown at the top of the table, the results of the statistical tests are shown underneath. Each hypothesis is summarised and tested for each map and for the combination of the results. In the raw data, all times are in milliseconds, the means and standard errors for **RRT** do not take the unsolved instances into account, whilst the medians do take the unsolved instances into account (a dash means that the median instance was unsolved).

<table>
<thead>
<tr>
<th></th>
<th>AR0308SR</th>
<th>Arena2</th>
<th>Door00d</th>
<th>Maze512-32</th>
<th>Orz100d</th>
<th>Plunderisle</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALT* time</td>
<td>14251</td>
<td>1893</td>
<td>25</td>
<td>86167</td>
<td>31536</td>
<td>21596</td>
<td>25912</td>
</tr>
<tr>
<td>AALT* time</td>
<td>182</td>
<td>54</td>
<td>5</td>
<td>891</td>
<td>354</td>
<td>230</td>
<td>286</td>
</tr>
<tr>
<td>RRT* time</td>
<td>22115</td>
<td>437</td>
<td>28</td>
<td>144592</td>
<td>6064</td>
<td>926</td>
<td>23848</td>
</tr>
<tr>
<td>ALT* duration</td>
<td>346.7</td>
<td>280.8</td>
<td>52.9</td>
<td>1460.7</td>
<td>630.3</td>
<td>452.7</td>
<td>537.4</td>
</tr>
<tr>
<td>AALT* duration</td>
<td>336.4</td>
<td>231.5</td>
<td>46.2</td>
<td>1424.3</td>
<td>664.7</td>
<td>424.8</td>
<td>521.3</td>
</tr>
<tr>
<td>RRT* duration</td>
<td>755.0</td>
<td>580.3</td>
<td>92.9</td>
<td>2168.5</td>
<td>891.0</td>
<td>1000.3</td>
<td>833.6</td>
</tr>
<tr>
<td><strong>Std. Error</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALT* time</td>
<td>1514.76</td>
<td>240.03</td>
<td>3.24</td>
<td>10265.36</td>
<td>3405.02</td>
<td>2421.46</td>
<td>42.44</td>
</tr>
<tr>
<td>AALT* time</td>
<td>10.73</td>
<td>4.89</td>
<td>0.53</td>
<td>90.80</td>
<td>28.12</td>
<td>14.58</td>
<td>24.83</td>
</tr>
<tr>
<td>RRT* time</td>
<td>12063.84</td>
<td>194.11</td>
<td>22.53</td>
<td>56917.30</td>
<td>5455.50</td>
<td>506.30</td>
<td>8295.44</td>
</tr>
<tr>
<td>ALT* duration</td>
<td>28.70</td>
<td>36.35</td>
<td>6.61</td>
<td>156.83</td>
<td>69.35</td>
<td>36.76</td>
<td>41.39</td>
</tr>
<tr>
<td>AALT* duration</td>
<td>31.07</td>
<td>27.58</td>
<td>5.26</td>
<td>1253.9</td>
<td>716.2</td>
<td>36.76</td>
<td>41.39</td>
</tr>
<tr>
<td>RRT* duration</td>
<td>77.31</td>
<td>104.72</td>
<td>11.35</td>
<td>3415.6</td>
<td>167.96</td>
<td>118.35</td>
<td>68.98</td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALT* time</td>
<td>11629</td>
<td>1570</td>
<td>39</td>
<td>71652</td>
<td>31830</td>
<td>16221</td>
<td>10619</td>
</tr>
<tr>
<td>AALT* time</td>
<td>171</td>
<td>47</td>
<td>5</td>
<td>703</td>
<td>339</td>
<td>215</td>
<td>178</td>
</tr>
<tr>
<td>RRT* time</td>
<td>41</td>
<td>29</td>
<td>2</td>
<td>125333</td>
<td>–</td>
<td>65</td>
<td>54</td>
</tr>
<tr>
<td>ALT* duration</td>
<td>319.2</td>
<td>208.5</td>
<td>38.5</td>
<td>1361.0</td>
<td>616.2</td>
<td>398.0</td>
<td>314.0</td>
</tr>
<tr>
<td>AALT* duration</td>
<td>299.9</td>
<td>195.4</td>
<td>37.5</td>
<td>1253.9</td>
<td>716.2</td>
<td>398.2</td>
<td>302.8</td>
</tr>
<tr>
<td>RRT* duration</td>
<td>685.9</td>
<td>338.7</td>
<td>70.1</td>
<td>3415.6</td>
<td>–</td>
<td>673.5</td>
<td>678.3</td>
</tr>
<tr>
<td><em><em>RRT</em> fail rate</em>*</td>
<td>0 / 40</td>
<td>2 / 40</td>
<td>0 / 40</td>
<td>14 / 40</td>
<td>22 / 40</td>
<td>0 / 40</td>
<td>38 / 240</td>
</tr>
</tbody>
</table>

#### (H1) AALT* has shorter runtime than ALT*.

<table>
<thead>
<tr>
<th></th>
<th>ALT* time</th>
<th>AALT* time</th>
<th>RRT* time</th>
<th>ALT* duration</th>
<th>AALT* duration</th>
<th>RRT* duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>5.511</td>
<td>5.511</td>
<td>5.281</td>
<td>5.511</td>
<td>5.511</td>
<td>13.368</td>
</tr>
<tr>
<td>p</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>r</td>
<td>0.871</td>
<td>0.871</td>
<td>0.835</td>
<td>0.871</td>
<td>0.871</td>
<td>0.863</td>
</tr>
<tr>
<td>Supported</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

#### (H2) AALT* has no worse quality than ALT*.

<table>
<thead>
<tr>
<th></th>
<th>ALT* time</th>
<th>AALT* time</th>
<th>RRT* time</th>
<th>ALT* duration</th>
<th>AALT* duration</th>
<th>RRT* duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0.511</td>
<td>1.661</td>
<td>1.772</td>
<td>0.000</td>
<td>3.387</td>
<td>0.175</td>
</tr>
<tr>
<td>p</td>
<td>0.295</td>
<td>0.045</td>
<td>0.039</td>
<td>&lt; 0.001</td>
<td>0.508</td>
<td>0.430</td>
</tr>
<tr>
<td>r</td>
<td>0.081</td>
<td>0.263</td>
<td>0.280</td>
<td>0.000</td>
<td>−0.536</td>
<td>−0.277</td>
</tr>
<tr>
<td>Supported</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

#### (H3) AALT* has shorter runtime than RRT*.

<table>
<thead>
<tr>
<th></th>
<th>ALT* time</th>
<th>AALT* time</th>
<th>RRT* time</th>
<th>ALT* duration</th>
<th>AALT* duration</th>
<th>RRT* duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>3.018</td>
<td>0.948</td>
<td>1.154</td>
<td>4.624</td>
<td>3.898</td>
<td>1.391</td>
</tr>
<tr>
<td>p</td>
<td>0.001</td>
<td>0.170</td>
<td>0.133</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>0.083</td>
</tr>
<tr>
<td>r</td>
<td>−0.477</td>
<td>0.150</td>
<td>−0.182</td>
<td>0.731</td>
<td>0.616</td>
<td>−0.220</td>
</tr>
<tr>
<td>Supported</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✓</td>
<td>✘</td>
<td>✓</td>
</tr>
</tbody>
</table>

#### (H4) AALT* has higher quality than RRT*.

<table>
<thead>
<tr>
<th></th>
<th>ALT* time</th>
<th>AALT* time</th>
<th>RRT* time</th>
<th>ALT* duration</th>
<th>AALT* duration</th>
<th>RRT* duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>5.269</td>
<td>5.511</td>
<td>5.027</td>
<td>5.511</td>
<td>5.444</td>
<td>5.377</td>
</tr>
<tr>
<td>p</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>r</td>
<td>0.833</td>
<td>0.871</td>
<td>0.795</td>
<td>0.871</td>
<td>0.861</td>
<td>0.850</td>
</tr>
<tr>
<td>Supported</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
ity of ALT*

H2 is supported across all benchmark maps except orz100d, and for the combined result. ALT* produces significantly higher quality solutions \((Z = 3.39, p < 0.001, r = -0.54)\) only for the orz100d benchmark map. We conjecture that because this map has features such as narrow corridors and unusually shaped obstacles, the abstract solution fails to correspond to an efficient ground solution.

The combined situation shows that the solutions produced by ALT* are only slightly better than those produced by AALT*, but far from a significant level and with very small effect size \((Z = 0.62, p = 0.263, r = -0.040)\).

Taking both H1 and H2 in conjunction, we can state with a high degree of certainty that AALT* is a more efficient planning algorithm than ALT* as it produces solutions of a comparable quality in significantly faster time.

(H3) AALT* finds solutions faster than RRT*

H3 provides the most mixed picture of any of the hypotheses. Before discussing the more detailed picture we note that the combined results demonstrate that AALT* is faster than RRT* to a statistically significant level \((Z = 2.245, p = 0.014, r = 0.15)\) but with a small effect size. However, looking at the results for each map individually, it is seen that H3 is only supported for two of the six maps (though in these cases the effect size is much larger than in the maps for which H3 is not supported). Despite the fact that AALT* finds solutions significantly faster than RRT* overall, the question remains about performance on the maps where H3 was not supported. This question will be looked at once H4 has been examined.

(H4) AALT* finds higher-quality solutions than RRT*

H4 is supported across all benchmark maps and also for the combined \((Z = 13.09, p < 0.001, r = 0.85)\) result. Across all benchmark maps and the combined results, the solutions are significantly higher quality with a large effect size. Taking both H3 and H4 in conjunction, there is a mixed picture. Whilst the solution quality of AALT* is universally higher than for RRT* over each map, the runtime is not universally better. This calls for further investigation in order to demonstrate if we can provide a stronger result about performance on these maps.

### 6.1 Further Comparisons with AALT* and RRT*

The RRT* algorithm is an anytime algorithm as the construction of the tree can continue beyond the point at which the first solution is found. We provide a further test to get more detail into the tradeoff between using ALT* and RRT*. This experiment is set up to allow RRT* at least as much time as AALT* requires to find its first solution (i.e. allowing RRT* to potentially find several improving solutions). We firstly run the AALT* algorithm on an instance to find a baseline time (call this \(t_{ALT}\)). We then use that time as a lower bound on the execution time of RRT*: if RRT* finds its first solution within \(t_{ALT}\) we allow it to continue constructing its tree until \(t_{ALT}\) has elapsed, when we accept the best solution found until that point.

This test is designed to show whether RRT* would produce better quality solutions if it spent the same amount of time searching as AALT*. If not, then it is reasonable to conclude that AALT* is a significantly more efficient planner (finds higher quality solutions in faster time) than RRT*. Therefore we test the following hypothesis:

(H5) AALT* finds higher-quality solutions than RRT*, even when RRT* is allowed the same time as AALT* as a lower-bound on its execution time.

We test the hypothesis on the same 40 benchmark instances per map used for the previous experiments. The results of the experiment are shown in Table 2. Even allowing RRT* the same amount of time as AALT* takes does not allow it to produce the same quality of solution. Overall H5 is supported \((Z = 12.21, p < 0.001, r = 0.79)\).

Given the evidence that AALT* produces higher quality solutions than RRT*, it is found to be significantly faster overall than RRT* we conclude that AALT* is superior to RRT* as a general purpose trajectory planner.

### 7 Conclusions

The main questions studied in this work are whether a graph abstraction approach can be generalised from a pure path-planning domain to the more realistic general trajectory planning problem, and whether the resulting algorithm provides performance competitive with the current best approaches. We have provided evidence supporting these claims. We have introduced the Abstract Augmented Lazy Theta* algorithm, which improves upon the Augmented Lazy Theta* algorithm (thus providing evidence that abstraction is indeed a useful approach in the trajectory planning domain). We have also demonstrated that Abstract Augmented Lazy Theta finds significantly higher quality plans than RRT* even when execution times are equalised (thus providing evidence that AALT* is competitive with current state of the art trajectory planning algorithms).
References


Nash, Alex, Sven Koenig, and Craig Tovey (2010). Lazy Theta*: Any-angle path planning and path length analysis in 3D. In: AAI.


