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 taken into consideration) *trajectory planning*.

Increasing the size of the search space (due to angle and speed being considered) 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* ($\mathbf{AALT}*$ 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 $\mathbf{ALT}*$).

We compare the performance of $\mathbf{AALT}*$ with both the $\mathbf{ALT}*$ and $\mathbf{RRT}*$ ($\mathbf{RRT}*$ 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 $\mathbf{AALT}*$ algorithm is significantly faster than $\mathbf{ALT}*$, whilst finding plans in comparable quality (supporting the idea that abstraction is an effective approach for trajectory planning). We also demonstrate that, although $\mathbf{RRT}*$ finds its first solution faster than $\mathbf{AALT}*$ on some maps, $\mathbf{AALT}*$ finds plans of significantly higher quality than $\mathbf{RRT}*$ even when $\mathbf{RRT}*$ 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* $p = \langle v_0, \ldots, v_k \rangle$ is a sequence of vertices $(v_0, \ldots, v_k \in V)$ 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 $p = \langle v_0, \ldots, v_g \rangle$.

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 ($p = \langle c_0, \ldots, c_k \rangle$) 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 $p = \langle c_0, \ldots, c_g \rangle$.

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 α 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-

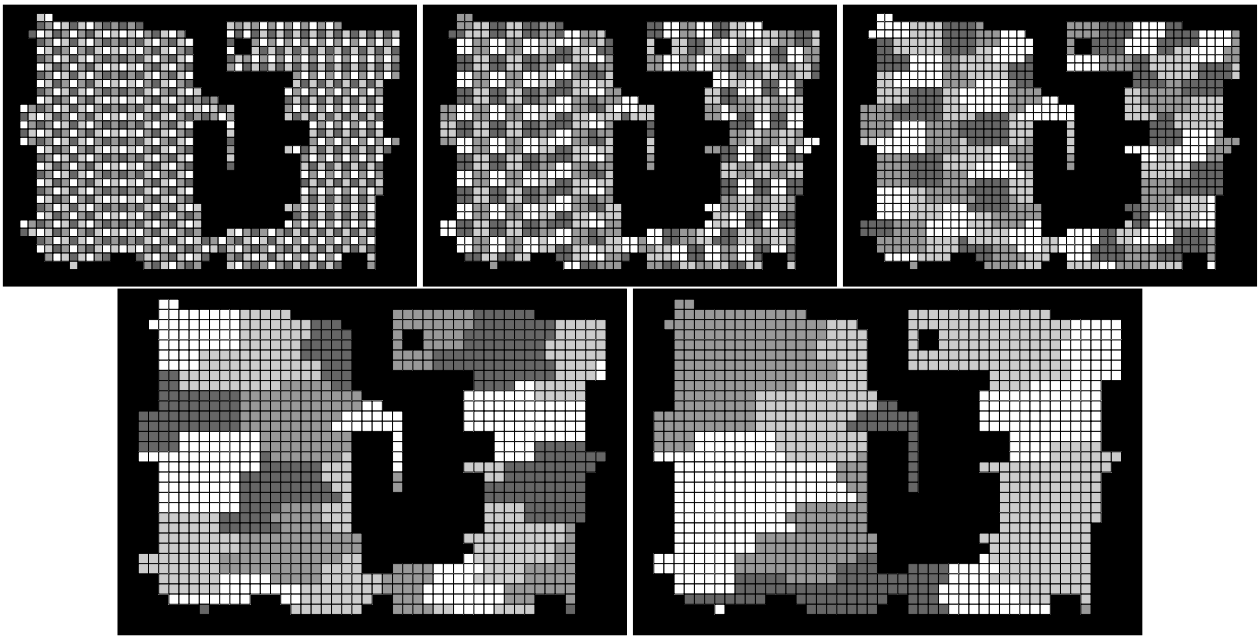


Figure 1: An example graph abstraction hierarchy on a grid-structured map. Nodes are shown as continuous regions of ground cells with the same shade. Each abstract node subsumes several nodes from the previous level in the hierarchy. The ground problem (top left) has approximately 1000 nodes whilst the 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 $LOS(parent(v),v)$ 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 $parent(v') = parent(v)$ and $parent(v) = v'$. 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., $||pos(c_{i+1}) - pos(c_i)||$. 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 (alternatively $parent(c_i)$ if the LOS check is successful) 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. $parent(c_i)$) to c_j. If the interval is disjoint with the current speed interval in c_i (resp. $parent(c_i)$) the SL check fails (c_j will not be

inserted to $open$ at this stage). Otherwise, the current speed interval in c_j is calculated and c_j is inserted into $open$. 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. $parent(c_i)$) 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 $open$ 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.

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 **RRT*** algorithm.

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. **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.

Algorithm 1 The Abstract Augmented Lazy Theta* Algorithm. Within this description, it is assumed that **ALT*** is an implementation of Augmented Lazy Theta* that can only explore a limited set of nodes (S_i in the description).

```
function AALT*(G,v_0,v_g)
    H = [h_0, ..., h_n] ← Abstraction Hierarchy of G
    R = [r_0, ..., r_n] ← Relaxed Solutions of H
    for i = 0 to n do
        S_i ← Ground states of r_i
        P ←ALT*(G,v_0,v_g,S_i)
        if P is a solution then
            return P
        end if
    end for
    return no solution
end function
```

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 a 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 *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 visted in the abstract plan. Failure of this

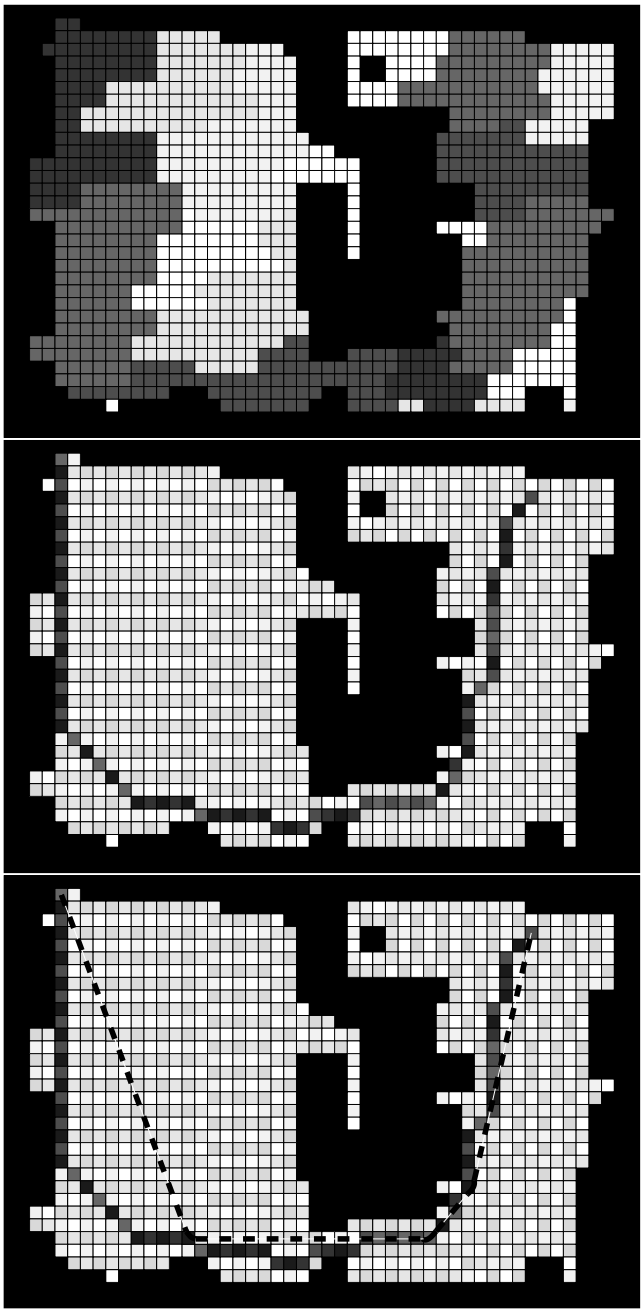


Figure 2: A presentation of the different stages in the $\mathbf{AALT}*$ algorithm. An abstract solution is shown in the top map (the abstract states which form the plan are displayed in darker colours and are roughly the states around the outside edge of the map). The ground relaxed solution is shown (middle) and the ground solution is shown in the bottom map, as the dashed line.

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 $\mathbf{ALT}*$ 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 $\mathbf{ALT}*$ 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 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 $\mathbf{AALT}*$ and so would have a longer duration.

5 Experimental Setup

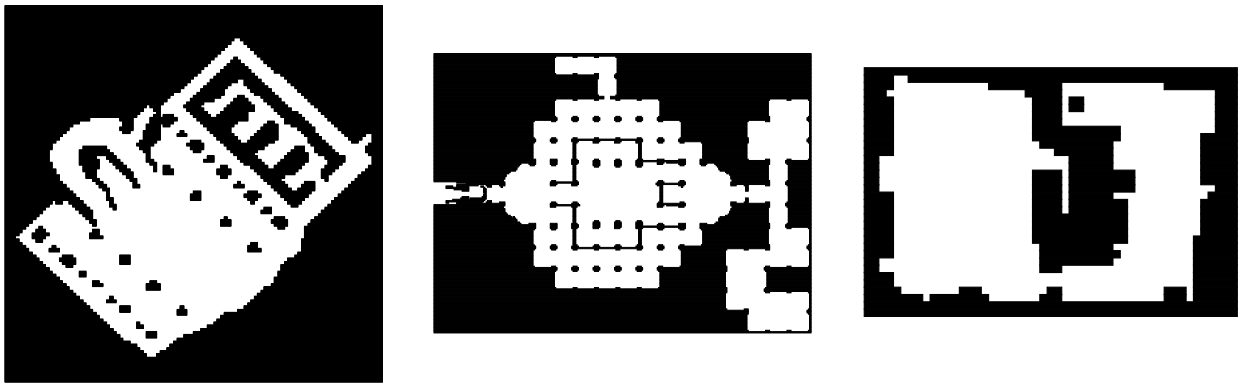
In order to demonstrate the effectiveness of $\mathbf{AALT}*$ in terms of its time efficiency and quality, we provide an empirical analysis comparing it with both the $\mathbf{ALT}*$ algorithm and the $\mathbf{RRT}*$ algorithm. Since $\mathbf{AALT}*$ is based upon the $\mathbf{ALT}*$ algorithm, it is important to demonstrate that it improves upon it. It is also important to show that $\mathbf{AALT}*$ performs competitively with current state-of-the-art algorithms (where $\mathbf{RRT}*$ 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 10×10 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 $\mathbf{RRT}*$ 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 $\mathbf{AALT}*$ (guaranteed to be unsolvable



(a) AR0308SR (b) Arena2 (c) Den009d

(d) Maze512-32-0 (e) Orz100d (f) PlunderIsle

Figure 3: The benchmark maps used for the evaluation. All are available from the MovingAI labs website.

as this tests reachability on the path planning version of the problem) are ignored as $\mathbf{RRT}*$ 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 perfomed 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 $\mathbf{RRT}*$: although any runs that reach this limit are implicitly recorded in the main results, we nevertheless report the failure rate of $\mathbf{RRT}*$ 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 finding 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 $\mathbf{AALT}*$. When r is negative, performance is degraded by using $\mathbf{AALT}*$. Finally we indicate whether the hypotheses were supported for each map and also when all results are combined.

(H1) $\mathbf{AALT}*$ finds solutions faster than $\mathbf{ALT}*$

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

(H2) $\mathbf{AALT}*$ finds solutions at least the same qual-

		AR0308SR	Arena2	Den009d	Maze512-32-0	Orz100d	PlunderIsle	Combined
Mean	**ALT**$*$ time	14251	1893	25	86167	31536	21596	25912
	AALT$*$ time	182	54	5	891	354	230	286
	RRT$*$ time	22115	437	28	144592	6064	926	23848
	ALT$*$ duration	346.7	280.8	52.9	1460.7	630.3	452.7	537.4
	AALT$*$ duration	336.4	231.5	46.2	1424.3	664.7	424.8	521.3
	RRT$*$ duration	755.0	580.3	92.9	2168.5	891.0	1000.3	833.6
Std. Error	**ALT**$*$ time	1514.76	240.03	3.24	10265.36	3405.02	2421.46	42.44
	AALT$*$ time	10.73	4.89	0.53	90.80	28.12	14.58	24.83
	RRT$*$ time	12063.84	194.11	22.53	56917.30	5455.50	506.30	8295.44
	ALT$*$ duration	28.70	36.35	6.61	162.92	71.73	38.15	2633.63
	AALT$*$ duration	31.07	27.58	5.26	156.83	69.35	36.76	41.39
	RRT$*$ duration	77.31	104.72	11.35	317.73	167.96	118.35	68.98
Median	**ALT**$*$ time	11629	1570	39	71652	31830	16221	10619
	AALT$*$ time	171	47	5	703	339	215	178
	RRT$*$ time	41	29	2	125333	–	65	54
	ALT$*$ duration	319.2	208.5	38.5	1361.0	616.2	398.0	314.0
	AALT$*$ duration	299.9	195.4	37.5	1253.9	716.2	398.2	302.8
	RRT$*$ duration	685.9	338.7	70.1	3415.6	–	673.5	678.3
	RRT$*$ fail rate	0 / 40	2 / 40	0 / 40	14 / 40	22 / 40	0 / 40	38 / 240
(H1)	**AALT**$*$ has shorter runtime than **ALT**$*$.							
	Z	5.511	5.511	5.281	5.511	5.511	5.511	13.368
	p	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
	r	0.871	0.871	0.835	0.871	0.871	0.871	0.863
	Supported	✓	✓	✓	✓	✓	✓	✓
(H2)	**AALT**$*$ has no worse quality than **ALT**$*$.							
	Z	0.511	1.661	1.772	0.000	3.387	0.175	0.615
	p	0.295	0.045	0.039	0.508	<0.001	0.430	0.263
	r	0.081	0.263	0.280	0.000	-0.536	-0.277	-0.040
	Supported	✓	✓	✓	✓	✘	✓	✓
(H3)	**AALT**$*$ has shorter runtime than **RRT**$*$.							
	Z	3.018	0.948	1.154	4.624	3.898	1.391	2.245
	p	0.001	0.170	0.133	< 0.001	<0.001	0.083	0.014
	r	-0.477	0.150	-0.182	0.731	0.616	-0.220	0.145
	Supported	✘	✘	✘	✓	✓	✘	✓
(H4)	**AALT**$*$ has higher quality than **RRT**$*$.							
	Z	5.269	5.511	5.027	5.511	5.444	5.377	13.09
	p	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
	r	0.833	0.871	0.795	0.871	0.861	0.850	0.845
	Supported	✓	✓	✓	✓	✓	✓	✓

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).

ity of $\mathbf{ALT}*$

H2 is supported across all benchmark maps except orz100d, and for the combined result. $\mathbf{ALT}*$ 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 $\mathbf{ALT}*$ are only slightly better than those produced by $\mathbf{AALT}*$, 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 $\mathbf{AALT}*$ is a more efficient planning algorithm than $\mathbf{ALT}*$ as it produces solutions of a comparable quality in significantly faster time.

(H3) $\mathbf{AALT}*$ finds solutions faster than $\mathbf{RRT}*$

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 $\mathbf{AALT}*$ is faster than $\mathbf{RRT}*$ 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 $\mathbf{AALT}*$ finds solutions significantly faster than $\mathbf{RRT}*$ 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) $\mathbf{AALT}*$ finds higher-quality solutions than $\mathbf{RRT}*$

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 $\mathbf{AALT}*$ is universally higher than for $\mathbf{RRT}*$ 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 $\mathbf{AALT}*$ and $\mathbf{RRT}*$

The $\mathbf{RRT}*$ 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 $\mathbf{AALT}*$ and $\mathbf{RRT}*$. This experiment is set up to allow $\mathbf{RRT}*$ at least as much time as $\mathbf{AALT}*$ requires to find its first solution (i.e. allowing $\mathbf{RRT}*$ to potentially find several improving solutions). We firstly run the $\mathbf{AALT}*$ 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 $\mathbf{RRT}*$: if $\mathbf{RRT}*$ 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.

	Z	p	r	Supported
AR0308SR	3.307	0.001	0.523	✓
Arena2	5.296	<0.001	0.837	✓
Den009d	5.000	<0.001	0.791	✓
Maze512-32-0	5.430	<0.001	0.859	✓
Orz100d	5.350	<0.001	0.850	✓
PlunderIsle	5.269	<0.001	0.833	✓
Combined	12.208	<0.001	0.788	✓

Table 2: Results of statistical testing for H5. H5 is the hypothesis that $\mathbf{AALT}*$ finds higher-quality solutions than $\mathbf{RRT}*$, even when $\mathbf{RRT}*$ is allowed the same time as $\mathbf{AALT}*$ as a lower-bound on its execution time.

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

(H5) $\mathbf{AALT}*$ finds higher-quality solutions than $\mathbf{RRT}*$, even when $\mathbf{RRT}*$ is allowed the same time as $\mathbf{AALT}*$ 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 $\mathbf{RRT}*$ the same amount of time as $\mathbf{AALT}*$ 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 $\mathbf{AALT}*$ produces higher quality solutions than $\mathbf{RRT}*$, is found to be significantly faster overall than $\mathbf{RRT}*$ we conclude that $\mathbf{AALT}*$ is superior to $\mathbf{RRT}*$ 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 $\mathbf{RRT}*$ even when execution times are equalised (thus providing evidence that $\mathbf{AALT}*$ is competitive with current state of the art trajectory planning algorithms).

References

Bäckström, Christer and Peter Jonsson (2013). Bridging the Gap Between Refinement and Heuristics in Abstraction. In: *IJCAI*, pp. 2261–2267.

Botea, Adi, M Müller, and Jonathan Schaeffer (2004). Near optimal hierarchical path-finding. In: *Journal of game development*, pp. 1–30.

Chrpa, Lukáš and Hugh Osborne (2013). Towards a Trajectory Planning Concept: Augmenting Path Planning Methods by Considering Speed Limit Constraints. In: *Journal of Intelligent and Robotic Systems*.

Daniel, Kenny, Alex Nash, Sven Koenig, and Ariel Felner (2010). Theta*: Any-Angle Path Planning on Grids. In: *J. Artif. Intell. Res. (JAIR)* 39, pp. 533–579.

Ferguson, Dave, Thomas M. Howard, and Maxim Likhachev (2009). Motion Planning in Urban Environments. In: *The DARPA Urban Challenge*, pp. 61–89.

Giunchiglia, Fausto and Toby Walsh (1992). A theory of abstraction. In: *Artificial Intelligence* 57, pp. 323–389.

Gregory, Peter, Derek Long, Craig McNulty, and Susanne M. Murphy (2011). Exploiting Path Refinement Abstraction in Domain Transition Graphs. In: *AAAI*, pp. 971 –976.

Howard, Thomas M. and Alonzo Kelly (2007). Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots. In: *International Journal of Robotic Research* 26.2, pp. 141–166.

Karaman, S, MR Walter, A Perez, E Frazzoli, and S Teller (2011). Anytime motion planning using the RRT*. In: *ICRA*, pp. 1478–1483.

Kavraki, LE, P Svestka, Jean-Claude Latombe, and MH Overmars (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In: *IEEE Transactions on Robotics and Automation* 12.4, pp. 566–580.

LaValle, S.M. and J.J. Kuffner (1999). Randomized kinodynamic planning. In: *ICRA*. Vol. 1. May. Ieee, pp. 473–479.

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

Phillips, Mike and Maxim Likhachev (2011). SIPP: Safe interval path planning for dynamic environments. In: *Proceedings of ICRA*, pp. 5628–5635.

Pivtoraiko, Mihail, Ross A. Knepper, and Alonzo Kelly (2009). Differentially constrained mobile robot motion planning in state lattices. In: *Journal of Field Robotics* 26.3, pp. 308–333.

Saitta, Lorenza and Jean-Daniel Zucker (2013). *Abstraction in Artificial Intelligence and Complex Systems*. Springer.

Seipp, Jendrik and Malte Helmert (2013). Counterexample-Guided Cartesian Abstraction Refinement. In: *ICAPS*, pp. 347–351.

Sturtevant, N. (2012). Benchmarks for Grid-Based Pathfinding. In: *Transactions on Computational Intelligence and AI in Games* 4.2, pp. 144 –148.

Sturtevant, Nathan and Renee Jansen (2007). An analysis of map-based abstraction and refinement. In: *SARA*, pp. 344–358.

Sturtevant, Nathan R and Robert Geisberger (2010). A Comparison of High-Level Approaches for Speeding Up Pathfinding. In: *AIIDE*, pp. 76–82.