Theoretical Overview for Geometric A* path-planning based approaches

Matteo Emanuele

January 2021

1 Introduction

Path finding algorithms are well needed in different fields: from robotics to videogames, the possibility to drive an agent from a point A to a point B in the space is crucial when resolving specific tasks.
In the following work we will see through path finding algorithms that have been studied in the research field. First we will take a brief look at the two most famous path finding algorithms in their standard form, A* and theta*.
After that, we will dive into the main topic of this report, which are declinations of the aforementioned algorithms: for our work we will focus our attention on Block A*, accelerated A* and Angle-Propagation Theta*.
We will take a look a their pseudocode, and at their results compared with the vanilla versions of the algorithms.

2 A* algorithm

A* (pronounced as "A star") is an algorithm that is widely used in pathfinding and graph traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph.
On a map, moving from a location A to a location B can be difficult, and sometimes blind-search algorithms based on the so call "cost function" solely are discouraging knowing that there are information in the task that could help us develop a procedure to solve it faster.
A* is able to do so, with the introduction of a function named Heuristic, that summed to the cost function allows us to define the so called "evaluation function".

$$f(n) = g(n) + h(n)$$

The evaluation function drive the search into the graph expanding the nodes that have the least value possible of f at each iteration.

One of the most important concepts bounded to the A* is that we can not choose any heuristic. We know that an heuristic to be as such must be admissible, which means that it never has to overestimate the cost to reach the goal.
Also, another important element is that the time complexity of A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the search space is a tree.

In the next page is reported a python based pseudo code of the A* algorithm *written by me.*

Algorithm 1: A* pseudocode.

```
def A*(starting_node, destination_node)
    closedlist = [ ]
    openlist = [ ]
    starting_node.f = starting_node.g = starting_node.h = 0
    openlist.append(starting_node) while openlist is not empty do
        Node_in_analysis = Retrive the node with the lowest f value from openlist
        remove Node_in_analysis from openlist
        add Node_in_analysis to closedlist
        if Node_ in_ analysis is destination_ node then
            return reversed path from starting_node to destination_node
        successors = Retrive succesors of Node_in_analysis
        for successor in successors do
            if successor is in closedlist then
                continue with the for loop
            successor.g = Node_in_analysis.g + g from Node_in_analysis to successor
            successor.h = heuristic value from successor to destination_node
            successor.f = successor.g + successor.h
            if successor is in openlist then
                if successor.g is higher than any other element.g in openlist then
                    continue to the new $for$ iteration
            append successor to openlist
        end
    end
    if nothing was returned then
        return no path was found from starting_node to destination_node
```

As we can see the algorithm requires to know where the agent will start and where we want to arrive. Without this information we can't obviously proceed.
Basically the main loop of the algorithm is based on checking each time, if we haven't arrived to the goal, the neighbors of the current node. We check their evaluation score f, and we pick the one that is the lowest between all the possible successor nodes, and we store it in a list to retrieve it when the task will end, to reconstruct the path through backtracking.

The complexity of A* for constant step cost functions is related to the error used to evaluate the heuristic: we can define $\epsilon = \frac{(h^* - h)}{h^*}$ to be the relative error of the heuristic, while $\Delta = h^* - h$ is the absolute error.
For constant step costs functions, we can affirm that the time complexity is $O(b^{\epsilon d})$, where d is the "depth" at which the goal state is located. As we can see, the branching factor is identified in the term b^{ϵ}. Intuitively, the time required from the algorithm to return a solution,if exist, it really makes sense that is linked to the branching factor of the problem togheter to the relative error of the heuristic, since such error express how good the heuristic is for the problem.

What are the weaknesses of A*? As it is largely discussed in the literature, A* presents a practical problems related to the memory: it's space complexity is exponential since it has to store all the nodes in memory during the search. Because of this drawback, time complexity is almost never really a problem for A*.
Different implementations has been provided to solve this issue, like the iterative deepening A*, which is a declination of the algorithm that use the iterative deepening concept to the heuristic search.

Geometric A*

Let's imagine that we want to use A* to move in a region of the space, populated by some obstacles. Plenty of applications like autonomous vehicles or non playing characters(NPC) in videogames requires to solve this path planning problem.
Whenever we are facing a problem of moving in an environment, we can try to schematize the surrounding areas of the agent with a grid: in such scenario, geometric version of search algorithms can be used to solve the path planning problem.
In such environment,everything has to comply to a grid representation. For example, an obstacle in the real world, would be represented in a grid world as in the picture that follows:

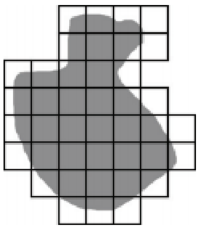


fig0: Obstacle representation in a grid world.
Picture from the paper "Any-Angle Path Planning for Computer Games"

We can talk of geometric A* whenever we are facing a path planning problem where the search space can be identified as the grid world in which our agent has to move from the start to the goal.
In such scenarios, one of the most common used heuristic is the so called "Manhattan distance", which is basically computing the number of movements the agent requires to go from the start to the end without moving diagonally.
When working on geometric A* we need to take into account the fact that the optimal solution in a grid world is usually not the same solution we would have found in a continuous representation world.

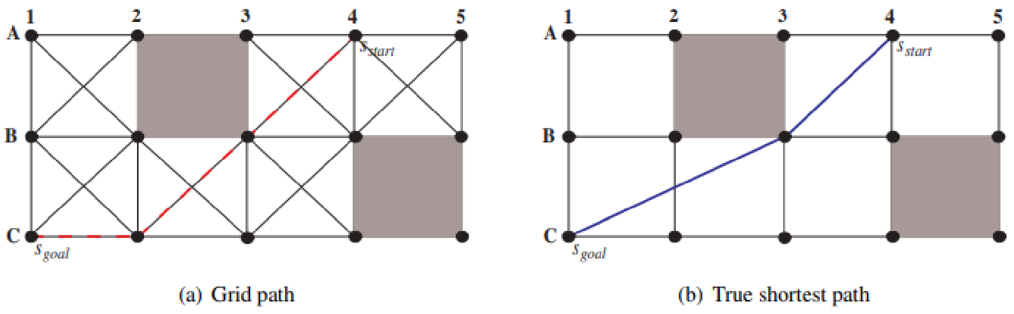


fig1: comparison of paths.
Picture from the paper "Theta: Any-Angle Path Planning on Grids"*

In general, we can affirm that the optimal path in a grid world, is a feasible path also in the real world, but usually is non optimal.
What problems may be faced in these kind of scenarios?
The turns that the agent usually performs are unnatural and sudden. In such cases we should take care of unsharp the turn elbow when working in scenario in which this sharp corners may result in a problem,whether is esthetic or practical.

Other practical problems that A* presents can be not strictly related to one of its modelling weaknesses, but can still lead to poor performances.
As it is stated in the paper *Applying Theta in modern game*, even though A* seems to works flawlessly in theory, when working on path finding solution may lead to terrible performance.

In the paper is discussed how in videogames like Age of Empires, where the world is modelled as a grid, A* generated terrible pathfinding solutions. In such genre of videogames the AI of the npcs requires to be as performative as possible in order to not ruin the gaming experience of the users, and back in the past this led to terrible opinion from the users towards the product.

Solutions partially better were later obtained with Theta*, an algorithm that was, in a certain sense, an improvement of A*.

3 Theta* algorithm

In this section we introduce the Theta* algorithm, an algorithm for path planning with any angle of travel.
One of the main conceptual differences with the geometric A* we just discussed, is that it propagates the planning information along the edges of the grid instead of the cells,exactly like visibility graphs. It basically combine the ideas of geometric A* with the idea of visibility graphs, where the change of directions occur only at the corners of the cells occupied by the obstacles.
The key difference when comparing theta* and A* is that in the latter we have predecessors and successors which are in a proximity relation between them: A* expand the search only towards the children of a state, which in the case of the vertex exploration would be the vertices that are nearby. While Theta* can drive the search from a vertex to any other vertices in the grid,is not bounded by a proximity relation.
But how does Theta* actually works? The path finding approach is very much related to A*: in fact, in terms of mere implementation, most of the code of Theta* is the same of A*. The key difference between them is an additional function for Theta* called in the pseudocode that will follow $Update\_Vertex$, which is applied at each iteration given that some conditions are satisfied. The conceptual difference is given by a simple reasoning: when finding the optimal path with A* in a grid, we usually face movements that are not optimal in a non grid world, as we already discussed.

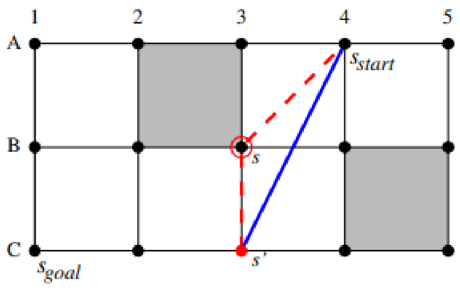


fig2: A path vs Theta* path.*
Picture from the paper "Theta: Any-Angle Path Planning on Grids"*

Imagine we want to move from the starting node showed in the fig 2 to the end node. Through geometric A* we would find a solution that is like the dotted red line. But common sense is telling us that there is no need for that sudden turn, we could just walk past like that line that goes from starting node to the ending node without passing for the middle node: Theta* implements exactly this kind of reasoning to optimize the path.
As we can see from fig 2, the constant blue line shows the path that Theta* would find. The idea behind the algorithm is the following: we apply A* in order to find the best path possible in a grid world, but we compute every time an additional check for the line of sight that goes from a node to another,and if the line of sight is free(which means that it does not walk through any obstacles in the world), then we can skip the middle nodes and go from the starting one to the latest one.

In the next page it is reported a python based pseudocode *written by me* for Theta* with the help of the pseudocode reported in the paper *"Applying Theta* in Modern Game"*

Algorithm 2: Theta* pseudocode.

```
def Theta*(starting_node, destination_node)
    closedlist = [ ]
    openlist = [ ]
    starting_node.f = starting_node.g = starting_node.h = 0
    starting_node.parent = starting_node
    while openlist is not empty do
        Node_in_analysis = Retrive the node with the lowest f value from openlist
        remove Node_in_analysis from openlist
        add Node_in_analysis to closedlist
        if Node_in_analysis is destination_node then
            return reversed path from starting_node to destination_node
        successors = Retrive successors of Node_in_analysis
        for successor in successors do
            if successor is in closedlist then
                continue with the for loop
            successor.g = Node_in_analysis.g + g from Node_in_analysis to successor
            successor.h = heuristic value from successor to destination_node
            successor.f = successor.g + successor.h
            if successor is in openlist then
                if successor.g is higher than any other element.g in openlist then
                    continue to the new for iteration
            Update_Vertex(Node_in_analysis,successor)
        end
    end
    if nothing was returned then
        return no path was found from starting_node to destination_node

def Update_Vertex(node, successor)
    if there is line of sight from node.parent to successor then
        if node.parent.g + distance(node.parent,successor) < successor.g then
            successor.g = node.parent.g + distance(node.parent,successor)
            successor.parent = node.parent
            if successor is in openlist then
                remove successor from openlist
            successor.g = successor.g + successor.h
            append successor to openlist
    else
        if node.g + distance(node,successor) < successor.g then
            successor.g = node.g + distance(node,successor)
            successor.parent = node
            if successor is in openlist then
                remove successor from openlist
            successor.g = successor.g + successor.h
            append successor to openlist
    end
```

As we can see, the majority of the algorithm implementation is exactly as it is for A*. The major difference, as we already stated, is on the function *Update_ Vertex*, that allows us to recompute the g value for the successor node. Note that the g value is recomputed in two different scenarios: if the parent of the node in analysis has line of sight with the successor node, or if the g value of the successor node is greater than the actual distance from the node in analysis plus the g value of the node in analysis.
This allows us to state that in the best case scenario, we are going to avoid a lot of sudden turns connecting the parents of a node to its successors directly, but also in the worst case scenario, we try to check if there is also a better path from the node to the successor.

After have analyzed the code it is clear why Theta* is an algorithm that tries to solve the "drunken-like" path that the basic A* generates.

What are the weaknesses of Theta*? Let's take a look at the following case:

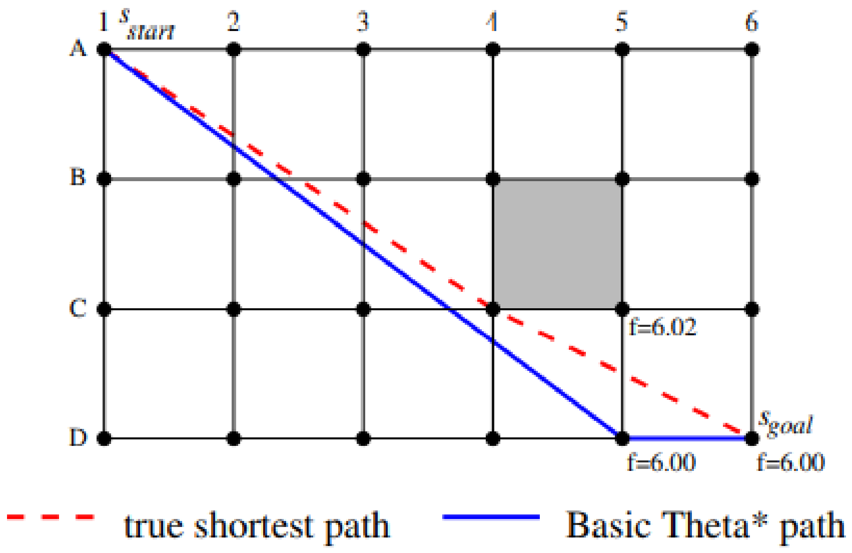


fig3: Theta problem.*
Picture from the paper "Theta: Any-Angle Path Planning on Grids"*

As we can see from above, there are situations in which Theta* presents the so called "heading changes". The reason behind this problem is quite straightforward: for how the algorithm is thought, when looking for the line of sight from a successor of a node and the parent of a node, everytime we will find this new path that connects those two since the last node added to the openlist is expanded before the second last, generating this problem. Thus, Theta* can be defined as *correct and complete*, but not optimal.
Another weakness of theta* is represented by its lack of optimality: the path returned by the algorithm is not guaranteed to be the shortest because the parent needs to be in line of sight with the successor everytime, which is not always a necessary condition for all the environments.

Introduction to different versions of A* and Theta*

In the next section of this paper, we will dive into different implementations and declinations of the aforementioned algorithms. A* more than any other algorithms works as the backbone for a whole family of path finding algorithms, as we have already saw with Theta*. We will take a look to three different algorithms:

Block A*;

Accelerated A*;

Angle-Propagation Theta*;

4 Block A*

Introduction

Path planning on maps is an important task as we have already discussed. In multiple applications the task can results in difficulties coming from the nature of the problem, like dynamically changing maps, a very common problem in strategic multiplayer videogames, for instance. In these scenarios, the algorithm has to compute the resulting optimal path receiving data from different players, everything in milliseconds. This state very clearly as we do care about computing the optimal path in the least amount of time possible.
Moreover grid worlds have limitations as we have already stated in the previous part of the paper. Theta* solves partially the problems that come up with such modelling of the world, but presents a significant downside with respect to A*, which is the time of computation. This should be of no surprise, since Theta* runs A* plus additional checks on each node, resulting in a slower algorithm.

In this section we present a new A* based algorithm called *Block A**, a Local Distance Database based algorithm that present higher performance respect to Theta* and A* in grid worlds.

The main idea behind is that A* time execution depends on the fact that it has to expand each cell every time it visits it. Block A* try to do the same, but not with a single cell every time, but with a block of n x n cells. We can look at the basic version of A* as the degenerative case of Block A* where the block is composed of 1 x 1 cells. The information between the cells inside the block are stored in a database and are retrieved when needed in order to expand the blocks in the most efficient way possible.

We will introduce here the main concepts that are needed to understand the in-depth steps of the algorithm.

Block: the group of cells we want to analyze at each iteration. The grid world has to be representable as a k x w grid where each cell is a square block n x n.

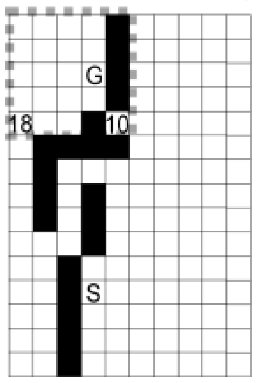


fig4: Block A grid world. The highlighted cell shows how the world is representable as 3x2 of those blocks. Picture from the paper "Block A*: Database-Driven Search with Applications in Any-angle Path-Planning"*

ingress cells/vertices: when we talk about a single block, we define ingress elements in two different(but related) ways depending if we are planning on vertices or on cells. The ingress cells for the block in analysis are those cells that have a different finite g value with respect to the last time the block was expanded. If we talk about ingress vertices, they are those vertices from the parent block that have a finite g value.

egress cells/vertices: when we talk about a single block we define egress elements in two different(but related) way, depending if we are planning on vertices or cells. The egress cells for the block in analysis are those cells that are unobstructed boundary cells adjacent to unobstructed cells in a neighbouring block. For vertices, egress vertices are all the vertices on the block boundary, simply speaking. This is because the block and its neighbouring blocks shares the same vertices for geometric construction.

LDDB: a local distance database, LDDB for short, is a data structure that stores the exact distance between the boundary points of a local region of the block in analysis. During the search of the optimal path the LDDB is queried to find the variation of the value g that the agent will see when enters a block at one location on the boundary cell of the block and exist from another boundary cell. The LDDB stores for every block couple the lowest path cost for each pair.

heap value: The priority of a block on the openlist. It's basically another way to call the evaluation score we used to decide which node to retrieve from the openlist of nodes in the basic A* implementation.

In a grid-based pathfinding problem, the search space is usually stored as a two-dimensional list or array, where each cell is represented with its coordinates. Each cell can be generally considered obstructed or not(1 or 0). Since our block generally has a n x n dimension, this means that we can have $2^{n \cdot n}$ number of permutation of the space for all the grid patters. This exponential growth may sound alarming, since we cannot pick the value of n to be too big. The incredible thing related to Block A* is that with a very small value of n(i.e. n=2, 2x2 block) this algorithm is already capable of outperform vanilla A*.
Also, for most of the environment, plenty of possible state can be eliminated since we need to consider only those pattern of the grid that allows the agent to move(i.e. a situation where the agent is complitely surrounded by obstacle blocking him makes no sense to be taken into account).
For the LDDB in this search space, with blocks of size n x n, there are at most 4(n - 1) ingress cells and 4(n - 1) 1 egress cells. Thus, the LDDB has a total of $2^{n \cdot n} \times 4(b\text{-}1) \times (4(b\text{-}1)\text{-}1)$ entries. The memory occupied by the LDDB can fits in under than 12MB, that can be cut up to 3MB if we consider rotational symmetries to reduce the patterns that require to be taken into account by the LDDB(rotational symmetries can be reduce the size up to a fourth of the initial size as it is stated in the paper *Block A*: Database-Driven Search with Applications in Any-angle Path-Planning*).

The algorithm

We will take a look to a conceptual schematization of the steps of the algorithms, and then we will comment the pseudocode.
The backbone of the algorithm, as it is intuitive, it's the same procedure we have in A*. The big difference here is the Expand function that is applied to a single block given its ingress cells/vertices, that actually allows to expand the block and compute the optimal path for that given block.

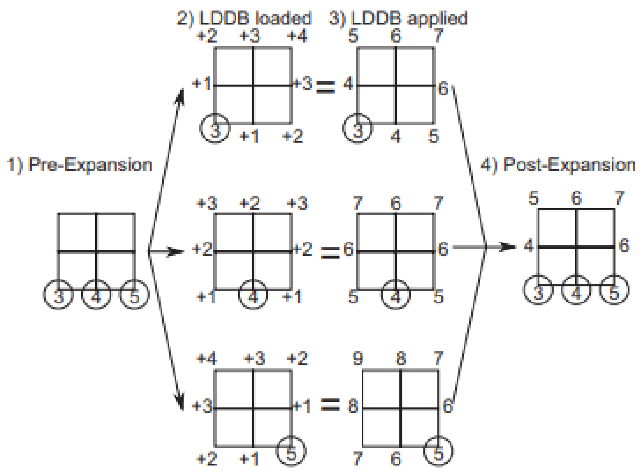


fig5: Expanding a block with block A using vertices for movement.*

Picture from the paper "Abstract: Block A and Any-Angle Path-Planning"*

For the following explanation of the four main conceptual steps of the Expand function, we will work with vertices as it is done in the *Abstract: Block A* and Any-Angle Path-Planning* paper. These steps are explained using as reference the figure 5 above.

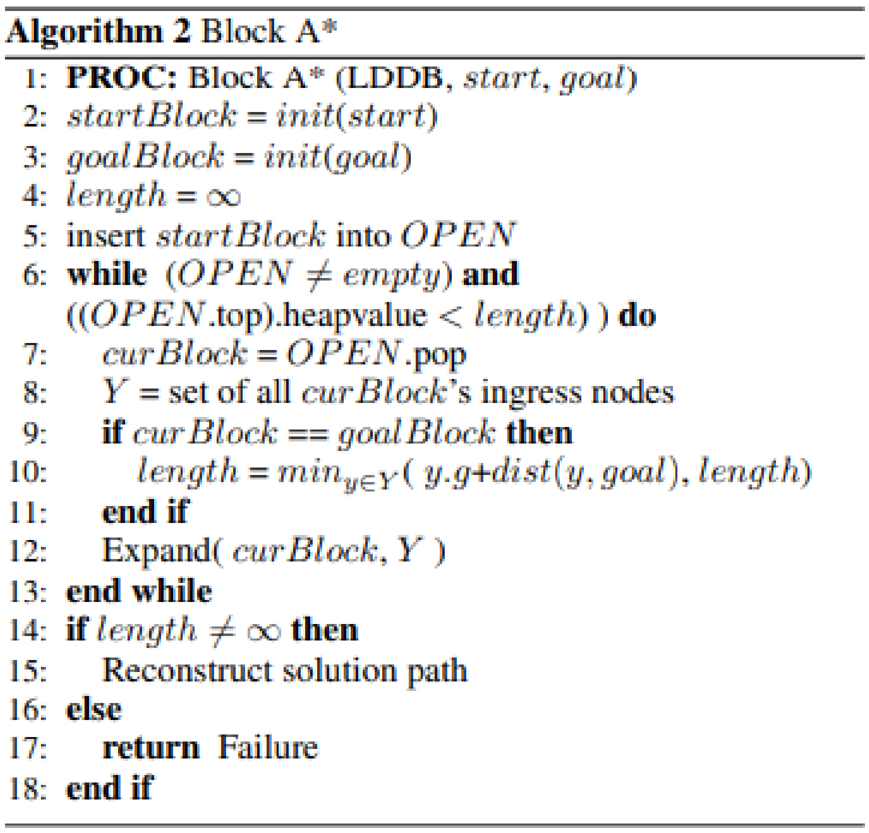
1) *Pre-Expansion:* Given a 2x2 block we first identify the ingress vertices in this case, which will be the three in the bottom part of the block circled. The egress vertices instead are all the vertices in the boundary(so basically all the vertices that there are, minus the one in the center). This is the pre-setup phase.

2) *LDDB loaded:* as we can see here we we have applied the LDDB value to each egress vertices. This is done for each ingress element separately, in fact we can clearly distinguish three scene in the fig5 above.
Let's notice how the value that were assigned to the egress vertices are in line with the definition of the LDDB.

3) *LDDB applied:* in this phase we add the g value of the ingress vertex that we are analyzing to the LDDB value retrieved for each vertices. This is done to find the shortest path from that ingress element to the i-th element.

4) *Post-expansion:* The block has been correctly expanded. The last step which is done in this last phase is to pick the final value of each egress element. This is done computing $min_{y \in Y}(x.g, y.g + LDDB(y, x))$, which mathematically represent the shortest path to go from x to y. For instance, for the top-center vertex the computation is done between three values, $min(6, 6, 8) = 6$.

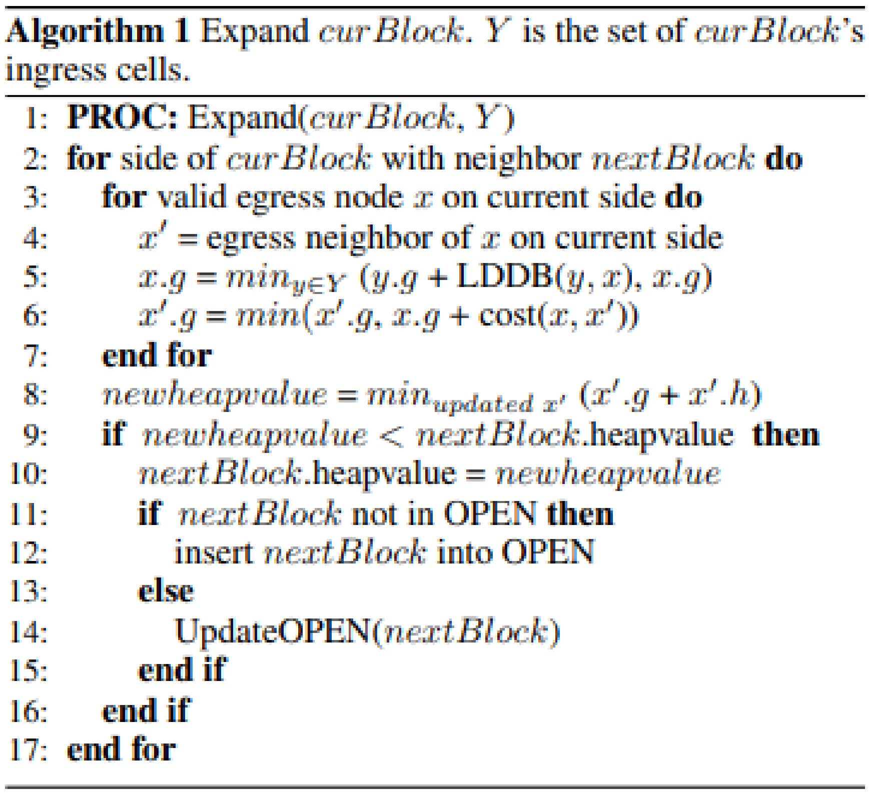
Let's take a look at the pseudo code now, *taken entirely from the paper of reference.*



Algorithm 2 Block A*

```
PROC: Block A* (LDDB, $start$, $goal$)
$startBlock = init(start)$
$goalBlock = init(goal)$
$length = \infty$
insert $startBlock$ into $OPEN$
while ($OPEN \neq empty$) and
((OPEN.top).heapvalue < $length$) ) do
    $curBlock = OPEN$.pop
    $Y$ = set of all $curBlock$'s ingress nodes
    if $curBlock == goalBlock$ then
        $length = min_{y \in Y}( y.g + dist(y, goal), length)$
    end if
    Expand( $curBlock$, $Y$ )
end while
if $length \neq \infty$ then
    Reconstruct solution path
else
    return Failure
end if
```

Pseudocode from the paper "Block A: Database-Driven Search with Applications in Any-angle Path-Planning"*



Algorithm 1 Expand $curBlock$. Y is the set of $curBlock$'s ingress cells.

```
1:  PROC: Expand($curBlock$, $Y$)
2:  for side of $curBlock$ with neighbor $nextBlock$ do
3:      for valid egress node $x$ on current side do
4:          $x'$ = egress neighbor of $x$ on current side
5:          $x.g = min_{y \in Y}$ ($y.g$ + LDDB($y$, $x$), $x.g$)
6:          $x'.g = min(x'.g, x.g$ + cost($x$, $x'$))
7:      end for
8:      $newheapvalue = min_{updated\ x'}$ ($x'.g + x'.h$)
9:      if $newheapvalue < nextBlock$.heapvalue then
10:         $nextBlock$.heapvalue = $newheapvalue$
11:         if $nextBlock$ not in OPEN then
12:             insert $nextBlock$ into OPEN
13:         else
14:             UpdateOPEN($nextBlock$)
15:         end if
16:     end if
17: end for
```

Pseudocode from the paper "Block A: Database-Driven Search with Applications in Any-angle Path-Planning"*

As we can see the algorithm is divided in two main blocks of code.
The first part is the main procedure for the Block A*: is very similar to the main procedure for A*, but we need to deal with some specific things related to the block-related logic of the algorithm. First of all, we will not initialize the starting and the goal cell, but we will initialize the starting and goal block. This initialization computes the shortest path from the starting/-goal cell to every reachable boundary cell within the block. There is a special exception for the case in which the starting block coincides with the ending block.
Basically the whole code proceed as in vanilla A*, until line 12 where the function Expand is called. That is the key difference.
As we can see, the Expand function is reported in the algorithm 1. Let's try to walk through the code of such function.
The first step is for sure to find all the egress cells, which we recall being the cells that are unobstructed and at the boundaries of the block, adjacent with unobstructed cells in the neighbors blocks.
The second step is to compute the g values. For each of these egress elements we will compute the g values using the data stored in the LDDB, but we change it with the computed one only if the new value is lower with respect to the previous one.
The final step to complete the expansion, we compute the heap value(the evaluation score), for each neighboring block, because we care on understanding which block we should move after this analysis, and if necessary we will insert it into the openlist or,if it's already there, just update its priority value.

Through these passages we can finally find an optimal path in a grid world. Here it is reported the steps of the algorithm:

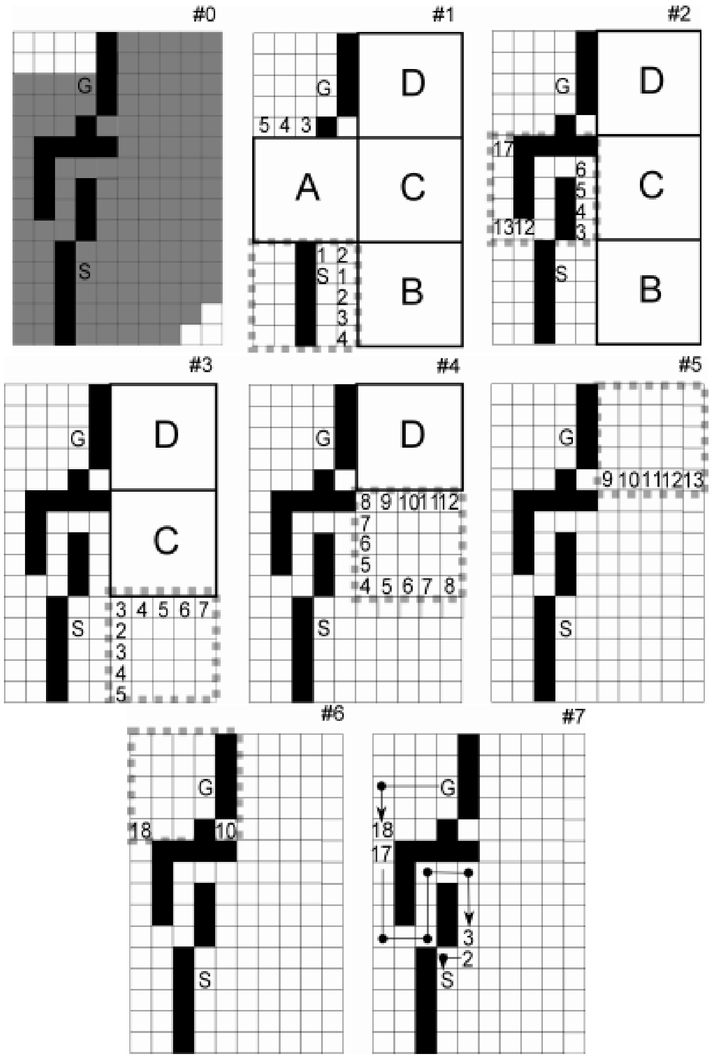


fig6: Block A example of pathfinding. Picture from the paper "Block A*: Database-Driven Search with Applications in Any-angle Path-Planning"*

In #0 we see the search space represented as a 15 x 10 cell grid. The shaded ones will be visited by A*. In this sophisticated example, the algorithm will visit most of the states. In #1 we can see the results of the initialization process we described previously. From this step we pass to #2 since block A is added firstly to the Openlist with an heap value of 11(A has only 1 egress cell from the starting block). Then B is added to openlist with an heap value of 13. A will be the block with the lowest heap value, thus will be expanded. From here we will add two different blocks to the Open list: the goal block will be added with a heap value of 23, the C block is added with an heap value of 13. So at this stage, we have three potentially expandable nodes in open list, but the two with the lowest heap value are B and C. Thus B will be expanded in this example(#3), and successively C (#4). In #5, we add D to the open list with an heap value of 13, while B is not reinserted in open since no g values of its cells improved. Finally we retrieve the goal block from the open list with its heap value of 23, which is telling us that the path we will find will be optimal and at least of length 23. in #7 we can see how the path is reconstructed at last.

Even though we introduced the terminology giving space also to the case in which the agent moves on the vertices rather than on the cells, the algorithm was written to work on the cells. The changes that are required to modify the algorithm on working on the vertices are quite simple. We just need to construct the LDDB taking the exterior vertices of the blocks as input rather than the boundary cells. As we have discussed in the definition of ingress and egress vertices, we know that the corner vertices, for instance, will be shared with other 3 blocks, things that does not happen with the cells. The reason why it's easy and non invasive to modify the algorithm to work on vertices it's because the block A* logic does not change, only the construction of the LDDB does. This version of the algorithm is called ***any-Angle block A****. Something that is worth being explicitly mentioned is that, giving how the algorithm reasons to decide which path to take inside a single block, thanks to the movement on the vertices we can achieve angles that are not bounded to be squared. In fact in the following example listed below we can see how the obtained path(the continuous line) results in passing in only three vertices to reach the goal from the start.

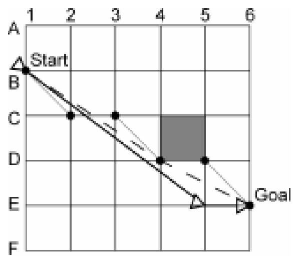
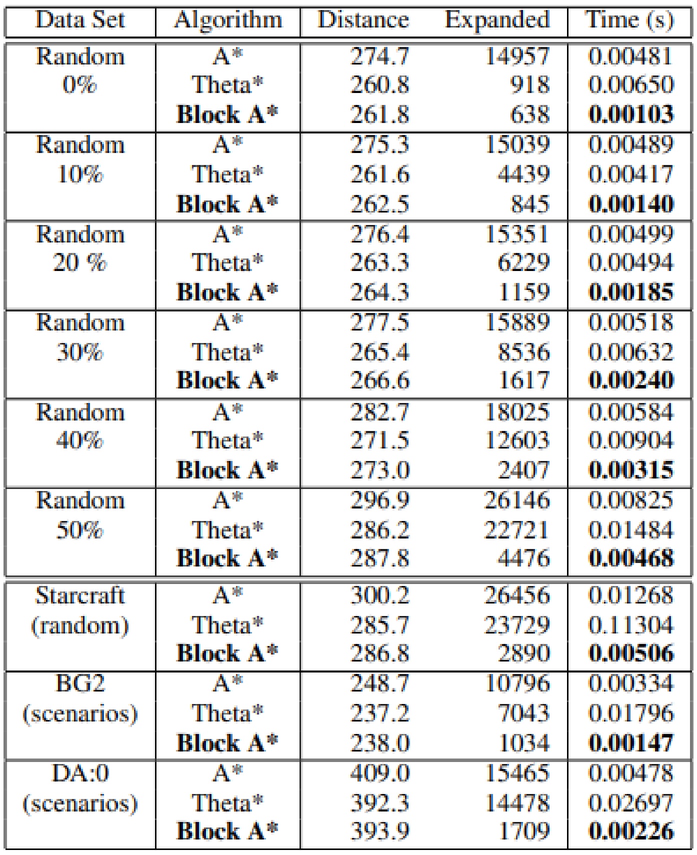


fig7: comparison A vs Block A* vs Theta*. Picture from the paper "Block A*: Database-Driven Search with Applications in Any-angle Path-Planning"*

In the picture is reported also the comparison with the path that the vanilla version of A* and Theta* will find. It is pretty clear in this case how block A* outperform both of the algorithm in terms of path quality. The path of Theta* is presented to be almost optimal, but it isn't, for the problems that were discussed in its own dedicated section.
Vanilla A* on the other hand present the classic suboptimality, very common with these kind of tasks. In this specific case, the basic version of the algorithm will find a zig-zag like path with changes at 90° resulting in many direction changes, pretty unnatural.

It is affirmable that Block A*, will always find the optimal path relative to the LDDB it uses.

It is reported here a comparison of results between the vanilla version of A*, Theta* and block A* in 9 different environment, where three of these are advanced scenarios from famous videogames(Dragon Age: Origins, Starcraft,Baldur's Gate).



Data Set	Algorithm	Distance	Expanded	Time (s)
Random 0%	A*	274.7	14957	0.00481
	Theta*	260.8	918	0.00650
	Block A*	261.8	638	**0.00103**
Random 10%	A*	275.3	15039	0.00489
	Theta*	261.6	4439	0.00417
	Block A*	262.5	845	**0.00140**
Random 20 %	A*	276.4	15351	0.00499
	Theta*	263.3	6229	0.00494
	Block A*	264.3	1159	**0.00185**
Random 30%	A*	277.5	15889	0.00518
	Theta*	265.4	8536	0.00632
	Block A*	266.6	1617	**0.00240**
Random 40%	A*	282.7	18025	0.00584
	Theta*	271.5	12603	0.00904
	Block A*	273.0	2407	**0.00315**
Random 50%	A*	296.9	26146	0.00825
	Theta*	286.2	22721	0.01484
	Block A*	287.8	4476	**0.00468**
Starcraft (random)	A*	300.2	26456	0.01268
	Theta*	285.7	23729	0.11304
	Block A*	286.8	2890	**0.00506**
BG2 (scenarios)	A*	248.7	10796	0.00334
	Theta*	237.2	7043	0.01796
	Block A*	238.0	1034	**0.00147**
DA:0 (scenarios)	A*	409.0	15465	0.00478
	Theta*	392.3	14478	0.02697
	Block A*	393.9	1709	**0.00226**

fig8: results comparison. Picture from the paper "Block A: Database-Driven Search with Applications in Any-angle Path-Planning"*

As we can see for small dummy environments with randomly generated obstacles, Theta* outperform A* since it expands fewer vertices. For more complicated environment we can see how Theta* lose its speed since it needs to expand as many nodes as A*, but having more computation inside the algorithm(the updateVertex function). Still. theta* finds way better paths than A* as we know from the theory. The problem on the computational time though, is crucial. Theta* tends to be, in general 10 times slower than A*, something that in some fields is not admissible: videogames require to compute paths in milliseconds, if not even less, and having slow algorithms risk to mess the entire performance of the game, experience included. Instead, talking about Block A*, we can see how it outperform both A* and Theta* and the reason behind it is that Block A*, thanks to the pre-computed results for the single block, can avoid potentially useless computations. On the other hand Theta* will have an expensive computation of a function every time, and if a lot of expansions were done, it starts to be very slow. Block A* basically presents performances that are 10 times better of Theta*, while having a path quality that is basically the same of Theta* also for very complex environments like Baldur's Gate levels, which are very known to be labyrinthic.

In conclusion, about Block A* we can affirm that it's a very efficient generalization of A*, thanks to his restricted search space. In the videogame industry A* is considered to be too slow for real time computation, a deal breaker when working on a videogame. Block A* is consistently faster than A* and Theta*, in both clogged and open areas. Theta* is much slower than A* in general, and when the heuristic is bad(open maps) both of them are unthinkable to be used. Block A* in the other hand, will always find shorter and more realistic paths compared to A*, paths comparable to the slower Theta* but with a computational time that is twice as faster as A* in its best performance, and 10 time faster than Theta* in average.

5 Accelerated A*

Introduction

In the following section is presented an alternative version of A*. So far we have seen the basic version of the A*, and then 2 different declination of it, Theta* and Block A*. Block A* is presented as an upgrade of the vanilla version,giving an enormous boost in the performance. The Accelerated A*(AA*) here presented finds its space of application to those domain in which we need to compute online optimal paths. As it is discussed in the paper, Accelerated A* is the path planning algorithm solution to those problems that are carried in 3D space while keeping into account also the time space. Since there are no accepted benchmark for 3D+time problems in the robotics or the aviation field, its performance will be evaluated on a grid world. It's performance will be compared to the vanilla version of A* and also to Theta*.

As it is in every path planning problem, the goal is to find the shortest path from a starting location to a destination.
We will discuss briefly the knowledge on top of which AA* is built. As we have widely discussed, A* presents plenty of problems in finding in practice the optimal path. When there are world representations in which is not possible to express powerful heuristics, suboptimality is achieved at best. Also, A* presents a *huge problem of memory* when working on gridworld: something that we haven't discussed in depth is that in a grid world is possible to achieve potentially more accurate path only if we have a high resolution of the grid. If we schematize the world with 10 cells,or if we use 1000 cells, the results will for sure change because in the second case we can try to solve the problem using narrow passages between the obstacles for istance. With both A* and Theta* this fragmentation of the grid world is not possible to be done as it pleases because each cell will be a node that has to be potentially explored and expanded by the algorithm. We know that the time and memory complexity of the algorithm increase with the depth of the solution and with the number of node to explore. This is, in fact, as we discussed, one of the reason why A* and theta* are not possible to be applied to plenty of different fields.
Accelerated A* algorithm uses a specific selection of the successor to be explored, without going for all the successors possible in order to reduce the dimension of the search space and optimize both time and memory requirement. The selection of the successors is what makes the algorithm, indeed, *accelerated*, since it speed up the search process reducing the time complexity from being linear in the number of nodes to being *constant* in the number of nodes. This is done choosing always a specific number of nodes,four, at each time.

The Algorithm

We will discuss the conceptual steps that are done by the AA* for achieving the optimal path, and then we will comment the pseudocode.
The majority of the algorithm will be the same of the A* and the Theta*, since it is a variation of the basic version, but for sure we can distinguish two phases that make the algorithm peculiar and different from its vanilla part.

Dynamic adaptive expansion: there will be a specific function in the algorithm that select four successor candidates, and only towards one of those four the search will proceed.

Progressive truncation: a truncation is applied to each generated successor in order to obtain the shortest path. It is similar conceptually to the path truncation done in Theta*, to avoid the passage over potentially useless nodes, but here is done checking with a specific reasoning the nodes stored in the closedlist.

We will now dive more into these two steps to understand more clearly how the algorithm works.
Given a starting vertex in which we are located in our grid world, we want to move ourself to a goal vertex. Taken our starting vertex, we will build the so called *"maximum unblocked square"* around the node, which basically consist,as the name suggest, on the biggest square that we can hipothetically draw around the node(placed at the center) without incorporate

any obstacle or obstructed cells. Another key feature that this square has to have, is that the goal node must fall at the most on the perimeter of the square, not inside.
Subsequently we consider as the candidates for the expansion the nodes that are the closest to the node in analysis, which is located at the center of the square. Being the "closest" means that the candidates will always be the one in the 4 cardinal direction with respect to the center of the square.
Here are reported two examples to understand how this successor are picked:

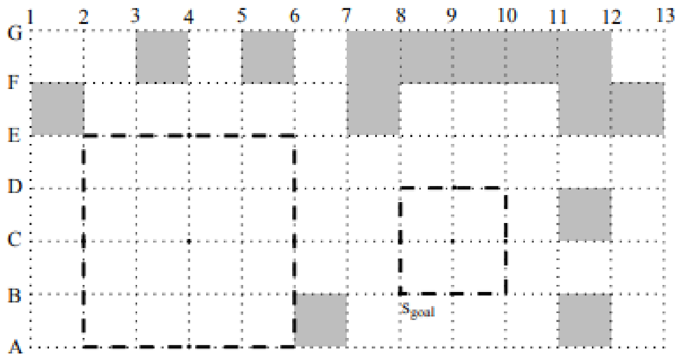


fig9: Block expansion in AA. Picture from the paper "Accelerated A* Trajectory Planning: Grid-based Path Planning Comparison"*

Here in the above picture we can see that the *the maximum unblocked block* for the node C4 is the biggest one in the left, the 4x4 block, while for the node C9 we have a smaller block, 2x2, and it's not the bigger square 4x4 E7-E11-B11-E7 because in that case the goal node will fall inside the perimeter, something that we have stated previously to be prohibited.
On the left most square, we can recognize the successors of C4(the center of the square) to be the 4 cardinal nodes, *successor*=E4,C6,A4,C2. We can be sure that these paths towards these nodes will always be the shortest and free because the length of the path to reach them will always be equal to half of the edge, and no obstacle will ever be in between for construction of the expansion process.
About the truncation of path that AA* applies, this happen taking into account *also the node already visited, which are the ones in the closedlist.* The potential parent that it's been looked for has to satisfy two requirements: it has to be connectable with the vertex, which means that the path must be obstacle-free, and passing through this vertex has to be the minimal g value.
It's important to point out that we do not have to traverse all the vertices in the closedlist again, since we can define a geometric method to chose which can be the candidates.
We can define an *ellipse test*: it will be an ellipse uniquely defined by the starting node, the node in analysis, and the current g value of the node in analysis.
The boundary of the ellipse are underlining those points that allow to reconstruct the minimum path from a starting node to a mid node to another general node, since it will always be at most equal to the distance to reach the goal: which means that whatever path is obtained through this truncation, is the best possible one. In the construction of such ellipse, the two nodes are placed in the two foci of the ellipse. Knowing their distance(the g value), we can find then the two radiuses, which formulas where been omitted but are reported in the paper.

Here is reported a graphical representation for more clarity:

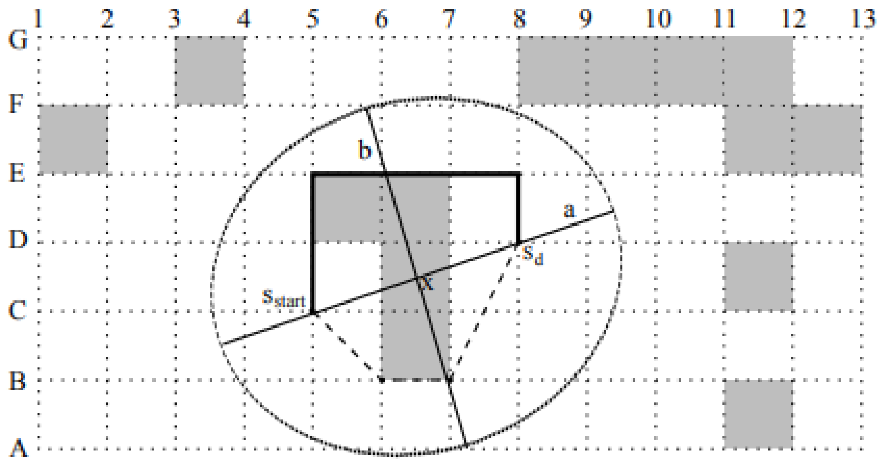
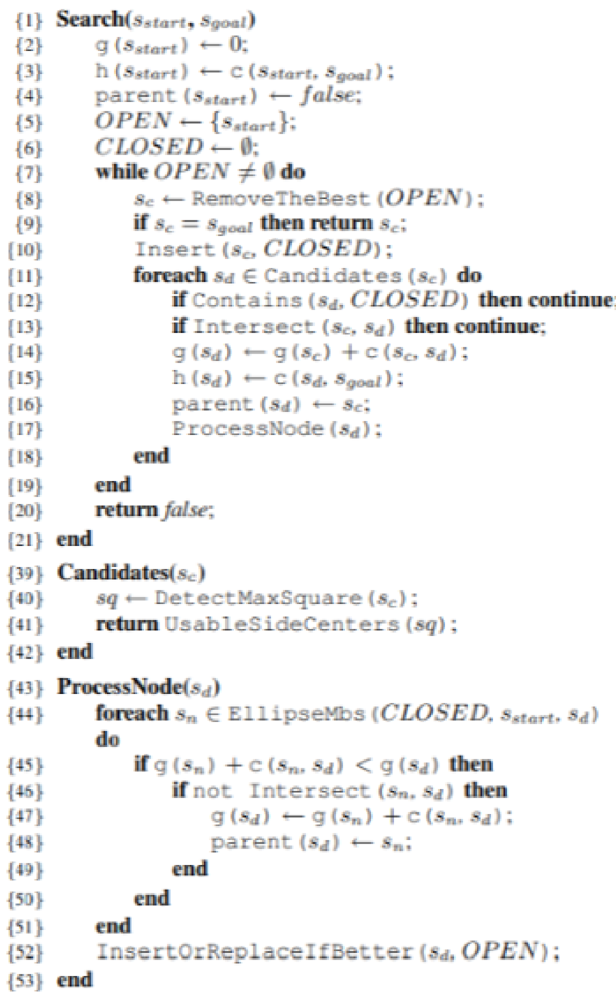


fig10: Ellipse test for AA. Picture from the paper "Accelerated A* Trajectory Planning: Grid-based Path Planning Comparison"*

This truncation indeed speed up the search in the closedlist because we will not search for every vertex in the list, but we will check only those vertices that are inside the closedlist and simultaneously inside the ellipse. In the above example, we can see how the path from s_{start} to s_d is truncated from the thick line with hard 90° turns, to the dotted line. The truncation brings to a result that is very similar, conceptually, to the one brought by Theta*, but the way is computed makes all the difference. As we can see the D8 vertex allowed to extract the vertex B7 from the closedlist and is assigned to be a new parent of D8, instead of E8. Then the node B6 is found as the parent of B7, allowing to ultimately reconstruct the dotted path which is indeed the shortest, truncating the starting path computed.
Something that makes sense to state clearly is that the reason why we limit the successor in closedlist within this ellipse region, is because the boundary of the ellipse are underlining those points in which moving with a straight line from the starting node to a mid node, plus the g value of a straight line path from that node to the node in analysis s_d is exactly the g value of s_d. What does this mean, in simple terms? It means that if we will think of going from the node towards a node that is outside of the ellipse, we cannot reach the s_d with a path that will be equal(or shorter) to the g value of the path we just found. This allows us to state that *the path we found through this ellipse test truncation is always the shortest, if exists.*
It is stated also in the paper how the retrivation of the nodes from the closedlist can be speeded up using an hashtable.

Here we report the pseudocode, *taken from the original paper:*



```
{1}  Search($s_{start}$, $s_{goal}$)
{2}      g ($s_{start}$) $\leftarrow$ 0;
{3}      h ($s_{start}$) $\leftarrow$ c ($s_{start}$, $s_{goal}$);
{4}      parent ($s_{start}$) $\leftarrow$ false;
{5}      $OPEN \leftarrow \{s_{start}\}$;
{6}      $CLOSED \leftarrow \emptyset$;
{7}      while $OPEN \neq \emptyset$ do
{8}          $s_c$ $\leftarrow$ RemoveTheBest ($OPEN$);
{9}          if $s_c = s_{goal}$ then return $s_c$;
{10}         Insert ($s_c$, $CLOSED$);
{11}         foreach $s_d \in$ Candidates ($s_c$) do
{12}             if Contains ($s_d$, $CLOSED$) then continue;
{13}             if Intersect ($s_c$, $s_d$) then continue;
{14}             g ($s_d$) $\leftarrow$ g ($s_c$) + c ($s_c$, $s_d$);
{15}             h ($s_d$) $\leftarrow$ c ($s_d$, $s_{goal}$);
{16}             parent ($s_d$) $\leftarrow$ $s_c$;
{17}             ProcessNode ($s_d$);
{18}         end
{19}     end
{20}     return false;
{21} end
{39} Candidates($s_c$)
{40}     $sq$ $\leftarrow$ DetectMaxSquare ($s_c$);
{41}     return UsableSideCenters ($sq$);
{42} end
{43} ProcessNode($s_d$)
{44}     foreach $s_n \in$ EllipseMbs ($CLOSED$, $s_{start}$, $s_d$)
         do
{45}         if g ($s_n$) + c ($s_n$, $s_d$) < g ($s_d$) then
{46}             if not Intersect ($s_n$, $s_d$) then
{47}                 g ($s_d$) $\leftarrow$ g ($s_n$) + c ($s_n$, $s_d$);
{48}                 parent ($s_d$) $\leftarrow$ $s_n$;
{49}             end
{50}         end
{51}     end
{52}     InsertOrReplaceIfBetter ($s_d$, $OPEN$);
{53} end
```

fig11: pseudocode for Accelerated A. Pseudocode from the paper "Accelerated A* Trajectory Planning: Grid-based Path Planning Comparison"*

We will now comment the pseudocode, recognizing in the line of codes the main passages that we have discussed before.
As we can see the main structure, as we have already said, is the same of the basic algorithm of A*. What changes here is that the candidates are extracted with a specific function.
In the function Candidates, as we can see, we are first computing the maximum unobstructed square around the given node, and then we are returning the *"Usable side centers"*, which correspond to the four nodes we mentioned before, the cardinal ones.
And for each of them we check if they are already in the Closed list of nodes(line 12) and if it is so, we skip at the next iteration of the for loop. We also check if there is an obstruction between the node in analysis and each of the candidates, and also here if this is true, we skip at the next iteration of the for.
For each of the candidates that arrives at line 14, we recompute their g value and their heuristic value, that in this case is the distance, and we also assign as parent to the new candidate with which we are working on the starting node that we are analyzing. We are doing so in order to have access to a reconstructable path through backtracking.
The last step is at line 17, where we apply the function *ProcessNode* to each of the candidates that didn't skipped their for loop iteration.
The processNode here is the moment in which we apply the *truncation.* Here we can see what we discussed earlier: for each successor inside the ellipse, for the given node s_d, we check if the evaluation score of the successor is lower than the g value of the node in analysis s_d, and if it is so we check if there are any intersection between such successor and the node in anal-

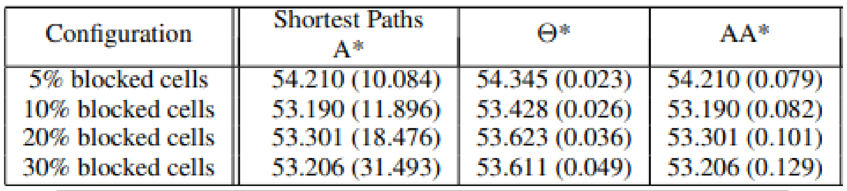
ysis. If the line of sight is free, we update the cost of the node s_d to be the evaluation value of the successor, and then we assign the successor to be the parent of the node in analysis. This assignment is what makes the truncation of the path happen effectively, because we are basically changing the branch value that will drive the backtracking towards this new branch. At the end of the processing of the node(line 52) we can see how we are adding/replacing(if better) the node to the openlist.

Let's point out few things before moving to the experimental results. We can see how the *ProcessNode* function is a function with a quadratic complexity: this may result to be heavy computationally talking, slowing down the algorithm performance. In the paper are briefly mentioned suggestion to partially solve this problem. Firstly, AA* seems to works faster than the vanilla version regardless of this quadratic complexity in the function, except for limit situations in which the time required for the AA* seems to reach very similar values to the one of basic A*. In general though the overall runtime of the algorithm is faster.
Another issue comes out from the quadratic complexity in the number of cells of the function *"DetectMaxSquare"*. Still also this issue can be addressed using the bisection method to accelerate the detection process. Reducing the number of tests for finding an unblocked path, the issue is partially addressed.

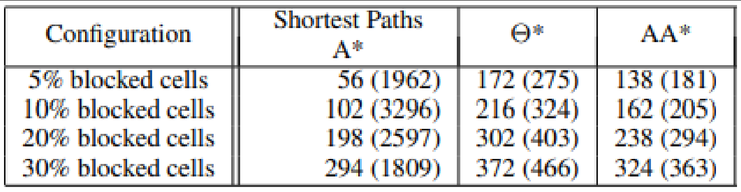
In this final part of this section dedicated to this algorithm, we comment the experimental results presented in the paper.

Experimental Results

We report here in two different tables the performances of the algorithms.



Configuration	Shortest Paths A*	Θ*	AA*
5% blocked cells	54.210 (10.084)	54.345 (0.023)	54.210 (0.079)
10% blocked cells	53.190 (11.896)	53.428 (0.026)	53.190 (0.082)
20% blocked cells	53.301 (18.476)	53.623 (0.036)	53.301 (0.101)
30% blocked cells	53.206 (31.493)	53.611 (0.049)	53.206 (0.129)



Configuration	Shortest Paths A*	Θ*	AA*
5% blocked cells	56 (1962)	172 (275)	138 (181)
10% blocked cells	102 (3296)	216 (324)	162 (205)
20% blocked cells	198 (2597)	302 (403)	238 (294)
30% blocked cells	294 (1809)	372 (466)	324 (363)

fig12: table results for Accelerated A. In the first one are reported the path lengths with the run time; in the second are reported the number of expansion and generated vertices. Tables from the paper "Accelerated A* Trajectory Planning: Grid-based Path Planning Comparison"*

The first set of experiments takes all places in randomized grids of size 100x100 cells. Since the world is randomly generated,usually this does not simulate real world scenarios.
The different rows of the table shows 4 level of densities of obstacles in the grid. Of course with the increment of the percentage of obstacles, it will increase the difficulty of the task, and the time required to solve it.
In table 1 we can see how the basic A* and the accelerated A* present the same path length at each test, but the difference in time execution is really notable. This is obviously because of the part that makes "accelerated" the algorithm, which is the brenching factor of 4 for the search.
As we can see in this randomized path A* and AA* presents the shortest paths, followed by Theta* with similar path length.
About the number of node expanded(with node expanded we mean the number of node visited in the while loop, so the number of nodes popped out from the openlist), we can see how AA* works with the least amount, which makes AA* the most optimized one in terms of memory among the three. This fits the theoretical analysis that we did earlier. We recall that the reason why this number remain low is because of the constant branching factor of 4.

Also here are reported another set up of environments were used as a testing ground. Instead of randomize a grid with a certain percentage of obstacles, specific configurations were handcrafted. Here are reported.

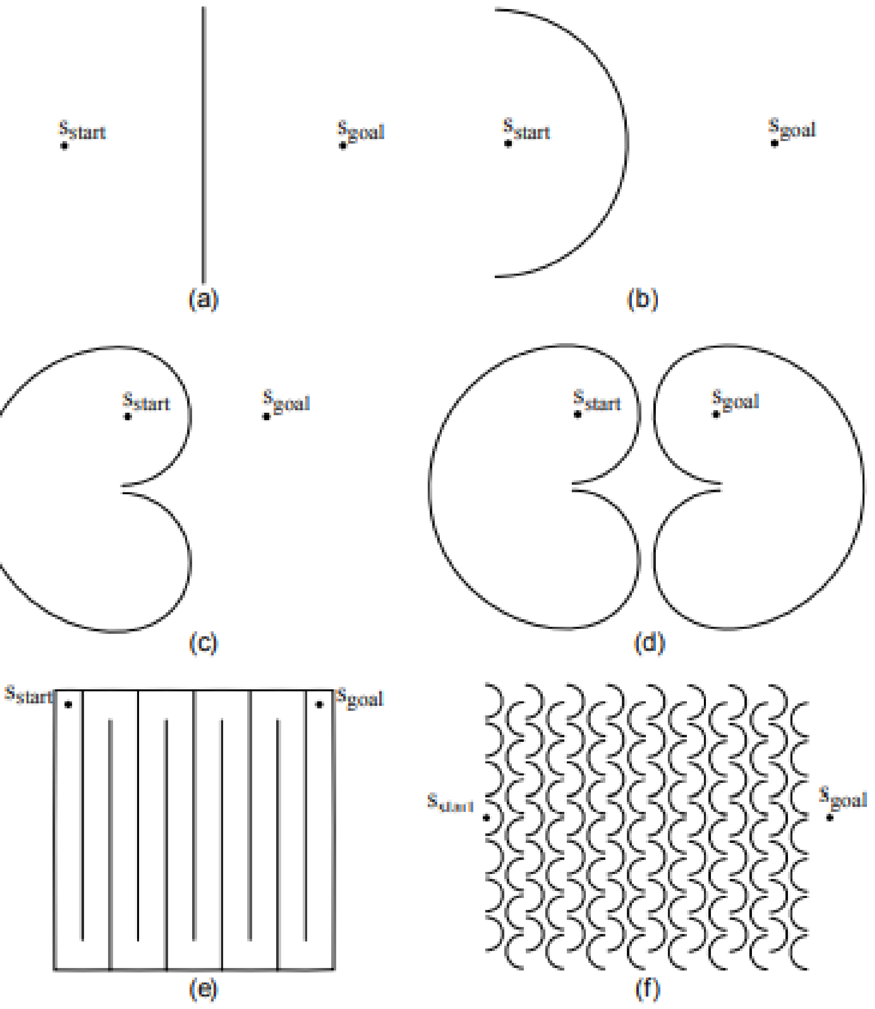
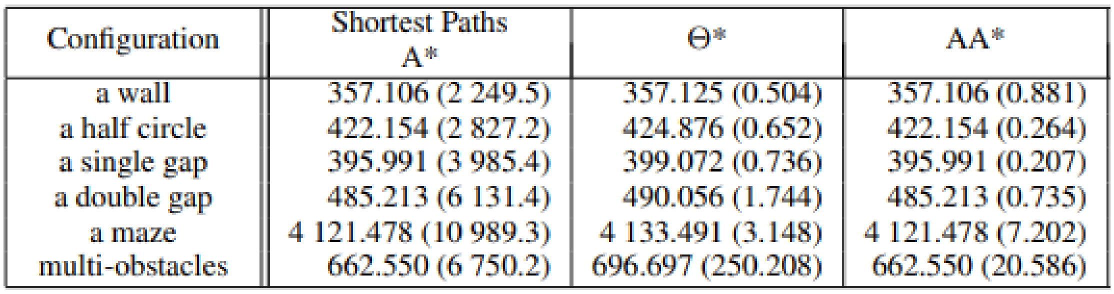
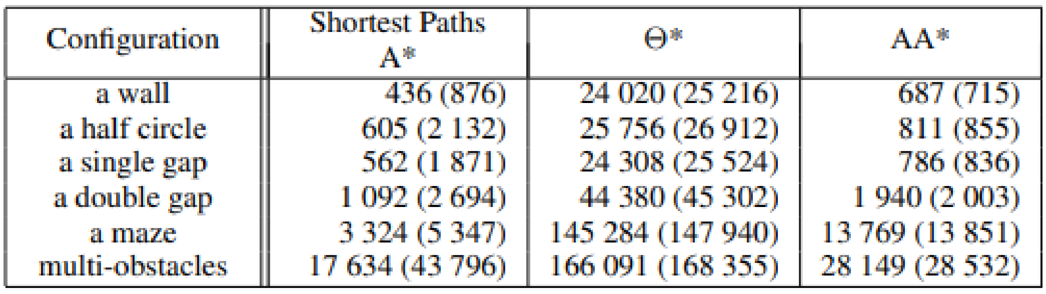


fig13: Testing environments. Picture from the paper "Accelerated A Trajectory Planning: Grid-based Path Planning Comparison"*

These testing environment can be distinguished(report directly the name from the paper): : (a) a wall, (b) a half circle, (c) a single gap, (d) a double gap, (e) a maze and (f) multi-obstacles. The start and the end node are pointed.
Configurations (a) and (b) have obstacles positioned so that they cause deviation from the preferred paths; in (b) the agent has to move in the opposite direction of the goal. Configurations (c) and (d) are used to test AA* on finding paths through small spaces. The (e) is used to test how "accelerated" the algorithm is, when the heuristic slows down the whole process, being inefficient. Lastly, (f) has a very complex environment, and it's a testing ground for "large-scale search problems.
In the next page are reported the experimental results inside two tables.



Configuration	Shortest Paths		
	A*	Θ*	AA*
a wall	357.106 (2 249.5)	357.125 (0.504)	357.106 (0.881)
a half circle	422.154 (2 827.2)	424.876 (0.652)	422.154 (0.264)
a single gap	395.991 (3 985.4)	399.072 (0.736)	395.991 (0.207)
a double gap	485.213 (6 131.4)	490.056 (1.744)	485.213 (0.735)
a maze	4 121.478 (10 989.3)	4 133.491 (3.148)	4 121.478 (7.202)
multi-obstacles	662.550 (6 750.2)	696.697 (250.208)	662.550 (20.586)



Configuration	Shortest Paths		
	A*	Θ*	AA*
a wall	436 (876)	24 020 (25 216)	687 (715)
a half circle	605 (2 132)	25 756 (26 912)	811 (855)
a single gap	562 (1 871)	24 308 (25 524)	786 (836)
a double gap	1 092 (2 694)	44 380 (45 302)	1 940 (2 003)
a maze	3 324 (5 347)	145 284 (147 940)	13 769 (13 851)
multi-obstacles	17 634 (43 796)	166 091 (168 355)	28 149 (28 532)

fig14: Table results for the second experiment. Tables from the paper "Accelerated A Trajectory Planning: Grid-based Path Planning Comparison"*

In these testing examples we can see how AA*, also here significantly reduce the number of expanded nodes in comparison to Theta* and to vanilla A*. What slows down the most the 2 latter algorithms is the necessity to extract the best candidate from the openlist everytime. A* results faster than Theta in a third of the tests, but also here AA* dominates in terms of speed, being more than 11 times faster than Theta*, and works with 15%-20% of Theta* expanded vertices.
AA* also perform wonderfully when facing path planning problems where the solution requires to pass through narrow passages(like the configuration (d)) , or also in (f), showing how even complex large-scale task can be addressed efficiently.

One last thing to point out is that the fact that AA* is an improved version for memory saving of the A* algorithm , it is not only a good thing in order to compensate the weaknesses of A*. The vanilla version of this algorithm presents criticality when facing complex task with a lot of nodes, and even when A* could potentially find a solution, in many problems may happen that it runs out of memory, leaving the task unsolved.
The fact that Accelerated A* is faster and more memory cheap, makes the algorithm an overall best solution choice since it is able to deal with more complex problems and actually *solves them.*

6 Angle-Propagation Theta*

Introduction

We will discuss here another algorithm that can be considered another declinated version of A*. To be more precise, we can consider this a declined version of the basic Theta*.
As we have discussed, theta* presents itself as an improvement of A*: being able to reduce the number of turns to allow the agent to make any kind of turn, and not only hard turn of 90° in a grid world, it returns generally more realistic and human-like paths. The downside is already clear: the computational time tends to be way higher, since it has to run a check on the visited nodes to control if it is possible to "prune" some unnecessary passages through some nodes.
The runtime of the basic Theta* can be linear in the number of nodes(since it has to check every cell expanded, and compute the line of sight for each cell). If we think it through, the runtime depends on the number of unexpanded visible nodes when expanding a vertex.
If we could be able to reduce somehow this time complexity, we would definately achieve an improved version of the basic Theta*: that is what Angle-Propagation Theta* aim to do.
AP Theta* reduce the runtime *per node expansion* of the basic Theta* from linear in the number of nodes to constant.
The key difference is that Theta* propagates its line of sight to check if the middle path is obstructed or not, while AP Theta* wants to propagate *angle ranges* and use them to check if we have line of sight between two vertices.
Let's bring a visual representation to make it more clear:

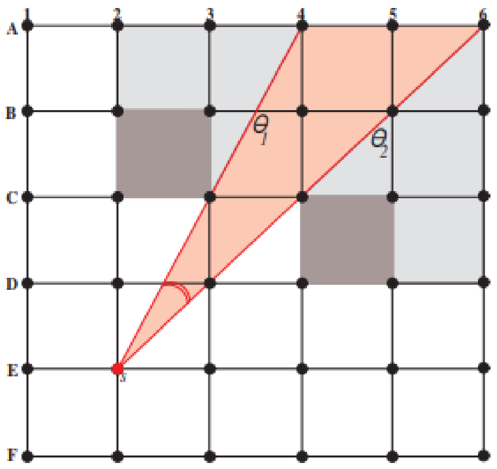


fig15: visual representation of AP Theta line of sight. Picture from the paper "Theta*: Any-Angle Path Planning on Grids"*

The angle propagation can be easily and intuitively explained with an analogy with the light. If there's a source of light in an environment, the way the shadowed region can be obtained is propagating the light beam from its source to the vertices of the obstacles in the direction of the cone of light.
If the light generates from the starting vertex where our agent lies, AP Theta*, intuitively, will not have any line of sight with those cells or those regions of the cells that are in the shadow projected by the obstacles, while will certainly have line of sight with those regions of the space that are in the cone of light that is able to propagates through.
Each contiguous region of points that have line-of-sight to the vertex can be characterized by two rays emanating from the vertex and thus by an angle range defined by two angle bounds θ_1, θ_2.
In the figure 15 we can see a region represented by a red triangle with one of the vertices located in the starting vertex: that region represent basically all the points bounded by θ_1, θ_2 that have line of sight with the starting location.

The algorithm

Here it is reported a python based pseudocode *written by me* for AP Theta* with the help of the pseudocode reported in the paper *"Theta*: Any-Angle Path Planning on Grids"*:

Algorithm 3: AP Theta* pseudocode.

```
def Theta*(starting_node, destination_node)
    closedlist = [ ]
    openlist = [ ]
    starting_node.f = starting_node.g = starting_node.h = 0
    starting_node.parent = starting_node
    while openlist is not empty do
        Node_in_analysis = Retrive the node with the lowest f value from openlist
        remove Node_in_analysis from openlist
        add Node_in_analysis to closedlist
        if Node_in_analysis is destination_node then
            return reversed path from starting_node to destination_node
        successors = Retrive successors of Node_in_analysis
        Update_Bounds(Node_in_analysis)
        for successor in successors do
            if successor is in closedlist then
                continue with the for loop
            successor.g = Node_in_analysis.g + g from Node_in_analysis to successor
            successor.h = heuristic value from successor to destination_node
            successor.f = successor.g + successor.h
            if successor is in openlist then
                if successor.g is higher than any other element.g in openlist then
                    continue to the new for iteration
            Update_Vertex(Node_in_analysis,successor)
        end
    end
    if nothing was returned then
        return no path was found from starting_node to destination_node

def Update_Vertex(node, successor)
    angleCheck = lb(Node_in_analysis)
          $\leq \Theta(Node\_in\_analysis, Node\_in\_analysis.parent, successor) \leq$
          ub(Node_in_analysis)
    if (Node_in_analysis is not starting_node) and (angleCheck) then
        /* path 1 */
        if node.parent.g + distance(node.parent,successor) < successor.g then
            successor.g = node.parent.g + distance(node.parent,successor)
            successor.parent = node.parent
            if successor is in openlist then
                remove successor from openlist
            successor.g = successor.g + successor.h
            append successor to openlist
    else
        /* path 2 */
        if node.g + distance(node,successor) < successor.g then
            successor.g = node.g + distance(node,successor)
            successor.parent = node
            if successor is in openlist then
                remove successor from openlist
            successor.g = successor.g + successor.h
            append successor to openlist
    end
```

```
def Update_Bounds(node)
    lb(node) = -inf
    ub(node) = +inf
    if node is not starting_node then
        for b in blocked cells adjacent to node do
            if for every s that belongs to corners(b): node.parent is s or
            ($\Theta$(node, node.parent, s)<0 or ($\Theta$(node, node.parent, s) = 0 and
              distance(node.parent, s) $\leq$ distance(node.parent, node)) then
                lb(node) = 0
            if for every s that belongs to corners(b): node.parent is s or
            ($\Theta$(node, node.parent, s)>0 or ($\Theta$(node, node.parent, s) =
              0 and distance(node.parent, s)$\leq$ distance(node.parent, node)) then
                ub(node) = 0
        end
        for every s in visible neighbours of node do
            if s is in closedlist and node.parent = s.parent and s $\neq$ starting_node
              then
                if lb(s) + $\Theta$(node, node.parent, s) $\leq$ 0 then
                    lb(node) = max(lb(node), lb(s) + $\Theta$(node, node.parent, s))
                if ub(s) + $\Theta$(node, node.parent, s) $\geq$ 0 then
                    ub(node) = min(ub(node), ub(s) + $\Theta$(node, node.parent, s))
            if distance(node.parent,s)< distance(node.parent,node) and node.parent is
              not s and (s is not in closedlist or node.parent is not s.parent) then
                if $\Theta$(node, node.parent, s) is negative then
                    lb(node) = max(lb(node), $\Theta$(node, node.parent, s))
                if $\Theta$(node, node.parent, s) is positive then
                    ub(node) = min(ub(node), $\Theta$(node, node.parent, s))
        end
```

How does AP Theta* works,intuitively? It proceed exactly like Theta*, checking for the possibility of using paths that bypass mid nodes through a check in the "update_vertex". This check though, get computed with the usage of θ_1 and θ_2: we will have basically two possible paths,like in basic Theta*, but here it is given by a condition that checks if the angle range is within two precomputed lower and upper bounds, and not through the simple line of sight. If it does, checking if the sum of the g value of the previous walked node and the distance between the previous walked node and the successor is smaller than the actual cost of the successor, we can skip the middle node, passing from the parent to the successor directly since *there is no obstruction.* This is a different approach for path choosing respect to Theta*, but the result is conceptually the same. If this check is not satisfied,we will opt for the passage through the mid node too, avoiding "shortcuts".

Given this intuition, we will briefly discuss the pseudocode.
The implementation of the code above apply exactly the intuition that we discussed. It's implementation is a bit tricky, and we will walk through the most important parts of the code to explain conceptually what is happening. Let's state what are some elements of the pseudocode that may sounds unfamiliar:

Θ: represent the concept of angle range more formally. Θ(s, p,s) $\in$ [90, 90]. This is the angle measured in degrees between the ray that start from vertex p to vertex s and the ray from vertex p to vertex s'. It is positive if the ray from vertex p to vertex s is clockwise from the ray from vertex p to vertex s'; zero if the ray from vertex p to vertex s has the same heading as the ray from vertex p to vertex s'; and negative if the ray from vertex p to vertex s is counterclockwise from the ray from vertex p to vertex s'.

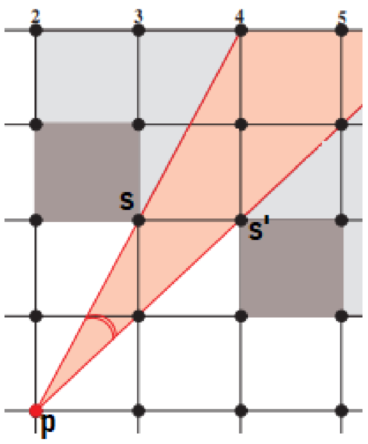


fig15: visual representation of the angle range. Picture from the paper "Theta: Any-Angle Path Planning on Grids"*

lb(node) and ub(node): lower bound and upper bound of the angle. AP Theta* maintains two additional values for every vertex. Togheter they form the angle range.

Firstly, it is pretty obvious that most of the code present the same structure of Theta*: the main difference is at line 13 of algorithm 3, where we call the function UpdateBounds. Indeed, the Update_vertex function is exactly identical except for the first if condition: we are no longer checking for the line of sight, so something will be different for sure here.
Our old concept of line of sight here is evolved in the condition called angleCheck(for readibility purpose): here we are checking everytime if the angle theta defined as stated above is within the lower and the upper bound of the vertex. In this case we are conceptually building the cone of visibility.

The main difference is obviously in the added function, UpdateBound. The name is self explanatory: this function wants to update accordingly to some conditions the upper and lower bound of the angle range. This is key to the path planning approach since we want to keep the bounds updated everytime we make the angle range propagates in order to understand if the agent can proceed to a certain node safely without encounter an obstructed vertex.
AP Theta* constrains the angle range of a given node based on every blocked cell b that is adjacent to the node in analysis.
The conditions stated in the two 'ifs' from line 6 to 11 are checking for two cases: basically in the first one we are checking if we need to set the lower bound to zero,and this is needed if we are going towards a negative angle range between the node,its parent,and the successor, since its range will span between 0 and 90°,positive values(line 6).
In the second case we are checking instead if the angle range is going to be positive between the node,its parent and its successor. In that case we are going to set the upper bound to zero,because th angle will be for sure negative in this case,moving its range between -90° and 0.(line 11).
It is interesting to comment what happen between line 20 and 24. Here we are basically in the case in which the distance between the parent of the node in analysis and the successor is less than the distance between the parent and the node itself, and also the successor hasn't been inserted in the closedlist, neither the node share a parent with the successor. In this odd and unfortunate case, we found ourselves without a solid guidance, and we cannot infere enough information about the successor. The most optimitstic thing that we can do, is to consider that there's line of sight between the node and its successor, so we are basically approaching this situation which isn't giving us enough information, as if we were moving with the basic Theta*.
we report here a visual representation of the four major steps of path planning of AP Theta*:

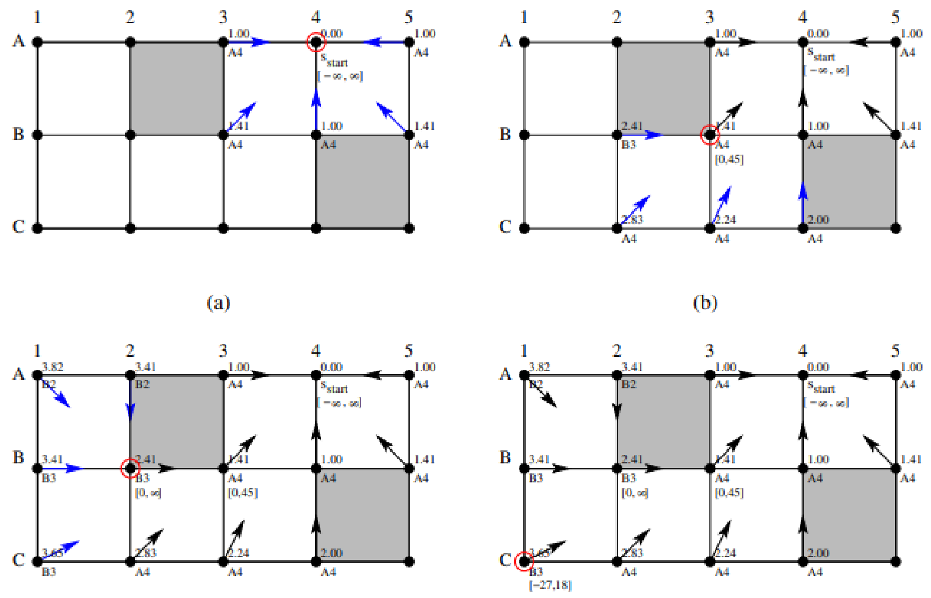


fig16: Path planning of AP Theta. Picture from the paper "Theta*: Any-Angle Path Planning on Grids"*

In this figure, the vertices are labeled with their g values and with arrows that points towards their parents. Red circle indicates the vertices that are being expanded, while the blues indicates vertices that are simply *generated* during the current expansion. The values in the square brackets are the angle ranges.
As we can see, the starting node is in in vertex A4. At the first step, from the start the upper and lower bound range of each vertex in the neighbors of the start are computed. The first choice for the path is A4. This is not given by the peculiarity of AP Theta*, but this node is chosen simply because it's the nearest to the goal. Then the next node in the path planning is expanded,and the successors are generated(blue arrows). Successively the path proceed on the vertex B3.Here the angle theta computes the angle range and pick the next successor as the new node in analysis(added to the openlist). The process is repeated again until it ends on the goal vertex that is C1. Also, we can notice how in the last steps the algorithm finds a sub-optimal path, passing for the vertex B2, and not pruning it. This is given by the angleCheck condition that actually overconstraints the path choice in the *update_vertex* function. The Θ angle range defined with the node in analysis B2, its successor C1(the goal) and its parent B3, does not allow the algorithm to connect directly in the path the successor and the parent because The Θ range does not fall between the upper and lower bound of B2. This is given by the*overconstraining led by the angle checks.*

How does AP Theta* performs compared to Theta*?
To answer this question we need to discuss the criticality of AP Theta*. Firstly, it is worth to say that, as Theta*, it is *correct and complete.* It does not always find best path,like Theta*, but will always return a valid solution to move from a vertex to another.
Also, the way on how theta constraints the angle ranges are very strict checks, that in some situation may overconstraint the path planning process, since the checks may fail sometimes. This lead to the ultimate assignment that we lastly discussed on the pseudocode,where AP Theta* still finds an unobstructed path if there exist any unblocked angle range, but the path founds in this scenarios thends to be usually longer than Basic Theta*.

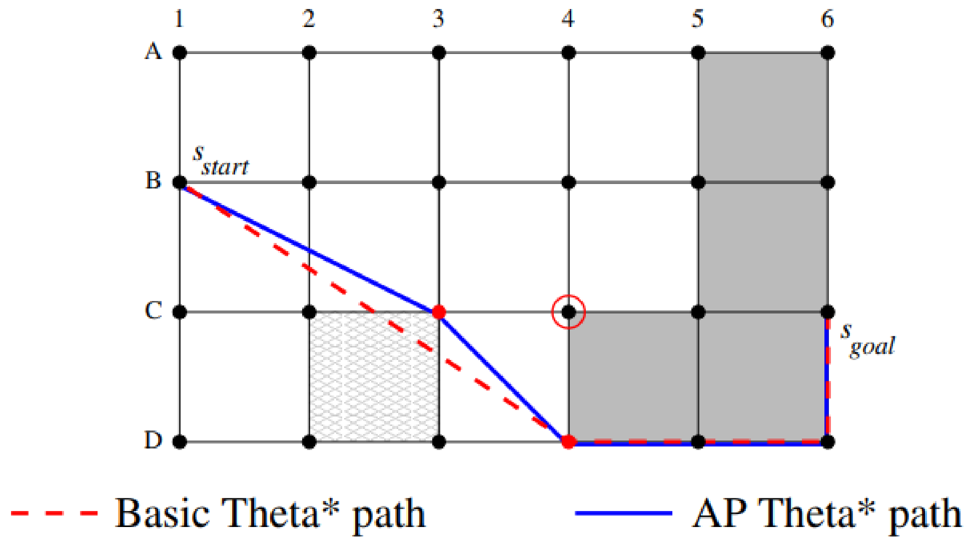


fig17: Path planning comparison: Theta vs AP Theta*. Picture from the paper "Theta*: Any-Angle Path Planning on Grids"*

In this case,for instance, the path found by AP Theta* is suboptimal, and of worse quality respect to basic Theta*. Let's try to discuss what is happening in this case.
AP Theta* starts from C4,expanding it with parent B1. It calculates the angle range of vertex C4. Vertex C3 is still unexpanded and not in the closed list. This is one of the key steps: since C3 is still not in the closedlist, AP Theta* doesn't have enough insight on C3 vertex. For instance, it doesn't know if the highlighted cell D2-D3-C2-C3 is blocked or not. Therefore, in this first step it is forced to chose a suboptimal path passing for the vertex on the perimeter of the cell, even though it is not obstructed. On the other hand, Theta* in this case doesn't have any problem,since it checks only for the line of sight between a node and its parent,thus doesn't find any abnormality on the highlighted cell.

This discussed scenario is important to understand that even though AP Theta* is a declination of basic Theta* and tends to work better in terms of time execution, its logical structure may lead to suboptimality in specific cases, resulting in longer paths.

Experimental Results

In this case we compare the basic Theta* with the Angle Propagation Theta*, adding also basic A* on grids as a reference.
The algorithms were tested on 6 different environments: 5 of these are random generated grid worlds, similar to the ones used for other algorithm evaluations, while one of the six is a game map. For this specific case, has been used Baldur's Gate 2 maps. Here a map sample is reported:

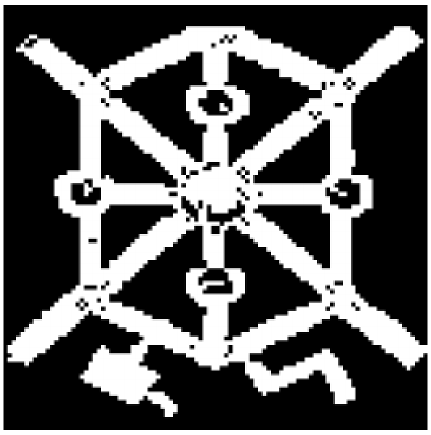
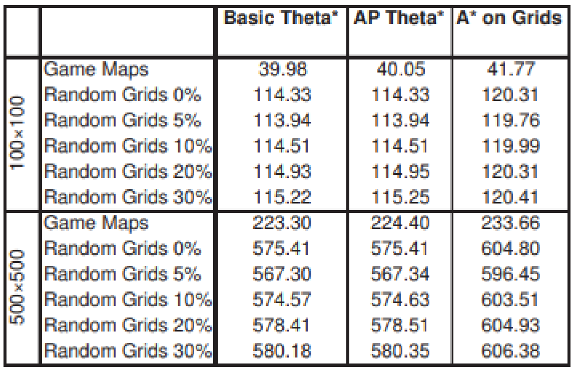
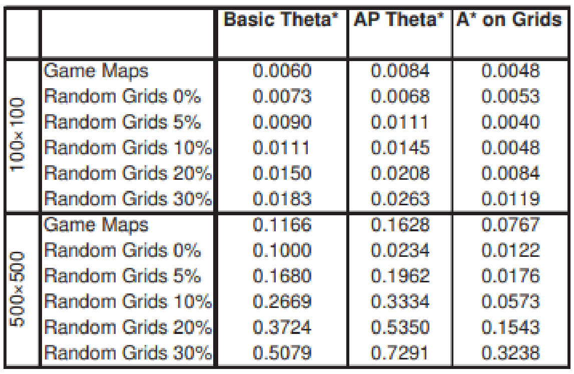


fig18: Baldur's Gate 2 map. Picture from the paper "Theta: Any-Angle Path Planning on Grids"*

In baldur's Gate 2 map, start and goal node were always placed in two(more or less) diametrically opposed locations of the map.
Here we report four different tables. The first one shows the path lengths, the second one is about the runtime, the third talks about the number of vertex expansions run by each algorithm and the last one shows the number of heading changes.
The results that will be shown in the following tables are averaged over 500 tries.

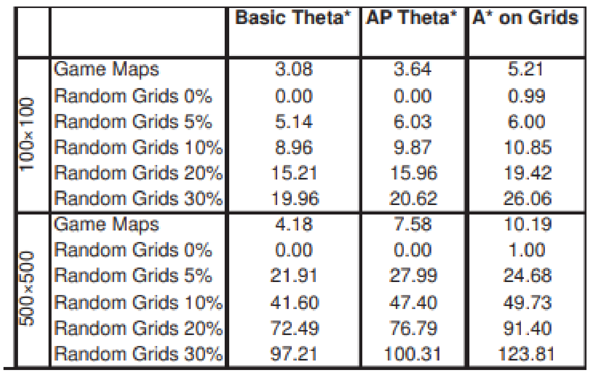


		Basic Theta*	AP Theta*	A* on Grids
100×100	Game Maps	39.98	40.05	41.77
	Random Grids 0%	114.33	114.33	120.31
	Random Grids 5%	113.94	113.94	119.76
	Random Grids 10%	114.51	114.51	119.99
	Random Grids 20%	114.93	114.95	120.31
	Random Grids 30%	115.22	115.25	120.41
500×500	Game Maps	223.30	224.40	233.66
	Random Grids 0%	575.41	575.41	604.80
	Random Grids 5%	567.30	567.34	596.45
	Random Grids 10%	574.57	574.63	603.51
	Random Grids 20%	578.41	578.51	604.93
	Random Grids 30%	580.18	580.35	606.38

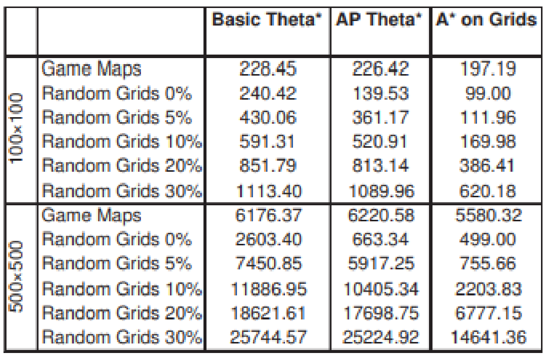


		Basic Theta*	AP Theta*	A* on Grids
100×100	Game Maps	0.0060	0.0084	0.0048
	Random Grids 0%	0.0073	0.0068	0.0053
	Random Grids 5%	0.0090	0.0111	0.0040
	Random Grids 10%	0.0111	0.0145	0.0048
	Random Grids 20%	0.0150	0.0208	0.0084
	Random Grids 30%	0.0183	0.0263	0.0119
500×500	Game Maps	0.1166	0.1628	0.0767
	Random Grids 0%	0.1000	0.0234	0.0122
	Random Grids 5%	0.1680	0.1962	0.0176
	Random Grids 10%	0.2669	0.3334	0.0573
	Random Grids 20%	0.3724	0.5350	0.1543
	Random Grids 30%	0.5079	0.7291	0.3238

fig19: Table results showing Path length and runtime. Tables from the paper "Theta: Any-Angle Path Planning on Grids"*



		Basic Theta*	AP Theta*	A* on Grids
100×100	Game Maps	3.08	3.64	5.21
	Random Grids 0%	0.00	0.00	0.99
	Random Grids 5%	5.14	6.03	6.00
	Random Grids 10%	8.96	9.87	10.85
	Random Grids 20%	15.21	15.96	19.42
	Random Grids 30%	19.96	20.62	26.06
500×500	Game Maps	4.18	7.58	10.19
	Random Grids 0%	0.00	0.00	1.00
	Random Grids 5%	21.91	27.99	24.68
	Random Grids 10%	41.60	47.40	49.73
	Random Grids 20%	72.49	76.79	91.40
	Random Grids 30%	97.21	100.31	123.81



		Basic Theta*	AP Theta*	A* on Grids
100×100	Game Maps	228.45	226.42	197.19
	Random Grids 0%	240.42	139.53	99.00
	Random Grids 5%	430.06	361.17	111.96
	Random Grids 10%	591.31	520.91	169.98
	Random Grids 20%	851.79	813.14	386.41
	Random Grids 30%	1113.40	1089.96	620.18
500×500	Game Maps	6176.37	6220.58	5580.32
	Random Grids 0%	2603.40	663.34	499.00
	Random Grids 5%	7450.85	5917.25	755.66
	Random Grids 10%	11886.95	10405.34	2203.83
	Random Grids 20%	18621.61	17698.75	6777.15
	Random Grids 30%	25744.57	25224.92	14641.36

fig20: Table results showing vertex expansion and number of turns. Tables from the paper "Theta: Any-Angle Path Planning on Grids"*

We will discuss here the key parts of these experimental results.

About run path length: we can see how A* results to have path lengths which are way larger than Theta* and AP Theta*. Basic Theta* seems to find the shortest path more often(70% of the time with respect to AP Theta*). The fact that AP Theta* path length is not that much bigger with respect to Theta* regardless of the fact that AP Theta* constrains the angle ranges more than necessary is showing how it is still a viable algorithm for path planning. Basic Theta* finds the true shortest path more often than the other algorithms.

About the run times: We can notice how A* seems to be the fastest among the three. This is no surprise since we have studied the theory behind the code: the Theta* versions of the algorithm are more computationally heavy. In fact, this is exactly the price to pay in order to have more realistic path when planning. In crescent order for runtime we have A*, followed by basic Theta* and for last AP Theta*.

About the number of vertex expansions: we can notice how A* , basic Theta* and AP Theta* is the crescent order of the number of vertex expansion. As it is obvious, with the increasing of the dimension of the map, the difference of expanded vertices between the algorithms become more obvious.
An interesting aspect about AP Theta* is that it reduce the runtime of Basic Theta* per vertex expansion, since we know that the runtime depending on the vertex expansion goes from linear(Basic Theta*) to constant(AP Theta*). As it is discussed in the paper, "*it is currently unknown whether or not constant time line-of-sight checks can be devised that make AP Theta* faster than Basic Theta**". Such shadowed narrows can be a good application for future research, since AP Theta* is known to be the first algorithm to have significantly reduced the runtime of any-angle path through a more complex check for obstructed cells(using the cone of visibility).

Lastly, about *the number of turns during the search:* still the ranking of the path planning algorithm here is the same of the previous metric: A*, Theta* and AP Theta*.
What we can clearly notice is how even though AP Theta* perform generally better in terms

of path length than A*, the basic Theta* algorithm results to be overall a better choice when deciding to path plan through a task. We do not have to forget what we have discussed in the past pages of this work: path planning problems require to be very frequently applied in very complex environments and needs to give an answer in terms of milliseconds, so the algorithms of Theta* does not seem to be a good choice for most of the applications. They find their place for task where the time of execution is not a key element for the success of the task. Also in the paper is stated clearly that since AP Theta* seems to perform generally more poorly respect to Theta*,for plenty of analysis discussed through the paper, the basic version of the algorithm were used. Also, as we have obviously noticed from the code, AP Theta* presents a way more problematic implementation, which discourage the usage if not for specific cases. AP Theta* is worth for an honorable mention because, as we have discussed before, it is the first significant step towards the reduction of the runtime for vertex expansion, and can be the first step towards a whole new algorithm design research field for path planning problems.

7 Ackowledgments

Thanks to Peter Yap, inventor of the Block A*, for the insightful emails exchanged during the creation of this theoretical report.

8 References

Hart, P., Nilsson, N., Raphael B.(1968). "A formal basis for the heuristic determination of minimum cost paths".

Stuart Russel, Peter Norvig(2009). "Artificial Intelligence: A modern Approach(third Edition)".

Peter Yap, Neil Burch, Robert C. Holte, Jonathan Schaeffer. "Any-Angle Path Planning for Computer games".

Kenny Daniel, Alex Nash, Sven Koenig(2010). "Theta: Any-Angle Path Planning on Grids".*

Peter Yap, Neil Burch, Rob Holte, Jonathan Schaeffer. "Block A: Database-Driven Search with Applications in Any-Angle Path-Planning".*

Phuc Tran Huu Le, Nguyen Tam Nguyen Truong, MinSu Kim, Wonshoup So, Jae Hak Jung (2016). "Applying Theta in Modern Game".*

David Sislak, Premysl Volf, Michal Pechoucek(2009). "Accelerated A Trajectory Planning: Grid-based Path Planning Comparison".*

Peter Yap, Neil Burch, Robert C. Holte, Jonathan Schaeffer(2011). " Abstract: Block A and Any-Angle Path-Planning".*