Master degree in Artificial Intelligence and Robotics

Planning and Reasoning

Any Angle Path Planning Algorithms

Submitted by:

Farooq Ahmad Wani

Matricola: 19476707

Accepted by:

Prof. Paolo Liberatore

Abstract

The most important problem in dealing with autonomous intelligent agents (*robots, agents in video games, etc*) is the navigation of the agent from the starting point to the goal point within the target environment. The target environment can be anything in which agent is performing the task. For example, if we consider the **Roomba** (*the cleaning robot*), the target environment is the area (*room*) which it needs to clean, the starting state of such robot is untidy room and the goal state of such robot is tidy room. Finding the non-obstruct path from the start point to the goal point is called **path planning**. The path planning problem is centric to many fields of Science and Technology. It is typically composed of two steps, the generation of the **graph problem**, by discretization of the continuous environment and then searching for a path by using the graph search algorithms. Roboticists or video game developers use many different techniques to find the optimal path within the discretized graph environment, but the most common ones are the traditional edge constrained pathfinding algorithms using the **A***. There are many drawbacks of these edge-constrained algorithms which lead to the design of the new group of algorithms called **Any Angle** algorithms.

Introduction

The "*edge constrained algorithms*" propagate the information along the edges of the graph. The main reason for using these algorithms is their **simplicity** (*very simple to understand*). These algorithms are efficient and provide a good trade-off between the runtime of the search and the path length. They possess the property of the **Generality**, i,e they can be used to search any Euclidean graph. The point which must be focused on, is that, while navigating the agent from start to goal coordinates, it should handle both path planning and the movement (*that is to take the path in such a manner that takes in account the dynamics of the system*). But it is very difficult to address both of these problems together, therefore, it is a common approach to treat them as separate problems. Here we will discuss only the **path planning** without discussing about the path tracking.

As stated above in **abstract,** the path planning process is done by the Discretization of the environment, then using the graph search algorithm the optimised path is returned. The discretization of the environment can be done in many ways and some of them are shown in **Figure 1:**

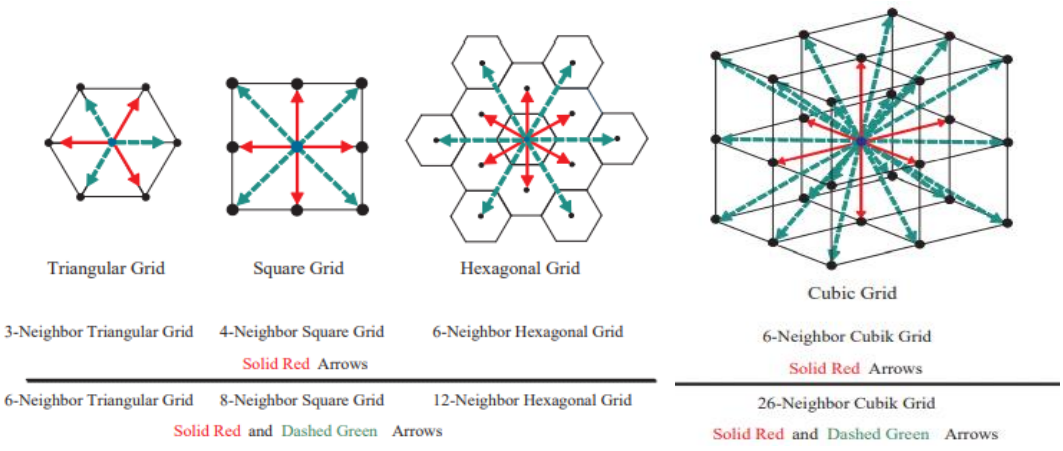


Figure 1

There are many types of graph generating algorithms, one such group of algorithms is edge constrained path planning algorithms. The main problem with the edge-constrained path planning algorithms is that path returned by them is not the shortest path. This mainly comes from the fact that these algorithms keep the resulted paths constrained to the edges. Therefore, we need the path planning algorithms that find the paths which are shorter than the shortest edge constrained paths, and should provide a good trade-off between the runtime of the search and the path length. **Any angle path planning algorithms** are variants of the heuristic path planning algorithms that transfer information along the edges but don't constraint the resulting paths to grid edges. Thus, the main difference between the two group of algorithms is the way they return the resulted paths, one can argue that why not use the post **smoothing techniques** in the paths resulting from the traditional edge constrained algorithms to make them equivalent to "any angle path algorithm" paths. It is because the process of smoothing the shortest grid paths (that is removing the unnecessary headings) doesn't change the topologies of the paths, therefore it doesn't have any effect on the way of circumnavigating the blocked grid cells. **Any Angle** path algorithms interleave the **A* search and smoothing**. The heading changes on **Any Angle** aren't artificially constrained to the specific angles. The term was coined by the **NASH (Nash 2007)**.

Any-Angle path planning algorithm aims to produce optimal or near-optimal solutions while taking less time than the traditional algorithms or nearly the same time as that of traditional grid-based solutions. As stated above, any angle algorithms are mainly based on the A*, they can be either **"Grid type A*"** based or **RRT (Rapidly Exploring Random Trees).**

RRT Any Angle algorithms will be discussed first followed by some of the "grid type A*" based Any Angle algorithms.

1. RRT Any Angle Algorithms for Path Planning

To understand the RRT algorithms, it is important to introduce some of the definitions and terminologies.

Robotic systems are expected to perform tasks in a **workspace W** that is often populated by physical objects, which represent an obstacle to their motion. For example, a robot working in a workshop must avoid collision with its static structures, as well as with other moving objects that may access it. Similarly, a mobile robot carrying baggage in an airport has to navigate among obstacles that may be fixed (*fittings, conveyor belts, construction elements*) or mobile (*passengers, workers*). Planning a motion then amounts to deciding which path the robot must follow in order to execute a transfer task from an initial to a final point without colliding with the obstacles. Clearly, one would like to endow the robot with the capability of autonomously planning its motion, starting from a high-level description of the task provided by the user and a geometric characterization of the workspace.

Developing automatic methods for planning is a very difficult endeavour. In fact, the spatial reasoning that humans instinctively use to move safely among obstacles has proven hard to replicate and codify in an algorithm that can be executed by a robot. To this date, planning is still an active topic of research, with contributions coming from different areas such as algorithm theory, computational geometry and automatic control.

In **Figure 2a** a **planar two arm** robot is shown. The robot moves in a Euclidean space **W** called a workspace. It is called a **planar robot** because its workspace is the plane of $\mathbf{R^2}$. The

workspace contains one obstacle shown as a circle in figure. The aim of the robot is to move the end effector (*shown by red dot*) of robot from the point A to the point B in the cartesian workspace $\mathbf{R^2}$ as shown in figure without colliding with the obstacle. The robot has two joints which correspond to two **degrees of freedom** i, e at any instant of time we can change only these two joint values (*angles*). If the length of links is known (*which are constant*) then at any point of time the end effector position can be known by knowing the value of these two joint angles (q_1 *and* q_2 *or alpha and beta*). This minimum number of parameters (*two here* q_1 *and* q_2) which can be used to represent every posture of the robot at any point of time is called the **configuration** of the Robot.

Shows the conversion of the workspace (W) of the robot manipulator with 2 degrees of the freedom to the configuration space.

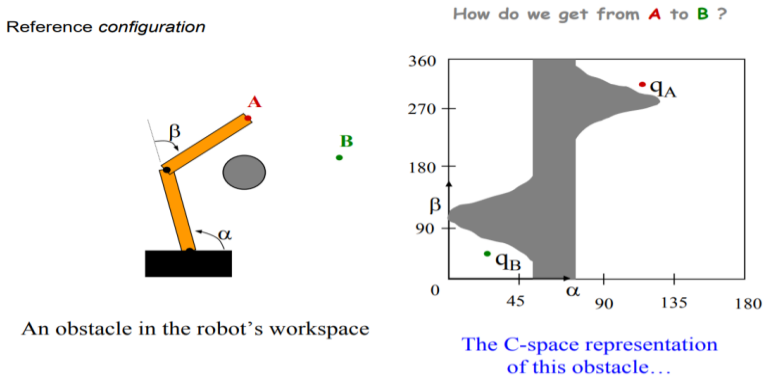
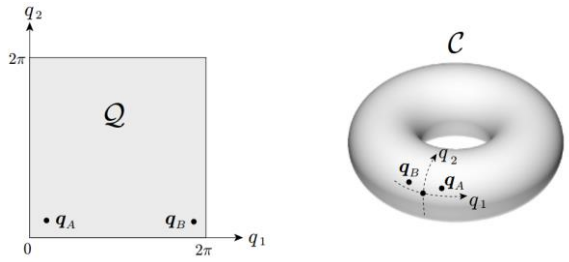


Figure 2a **Figure 2b**



The configuration space of a two-arm planar robot; left: a locally valid representation as a subset of R^2*, right: a topologically correct representation as a two-dimensional torus*

Figure 2c

A very effective scheme for planning is obtained by representing the robot as a mobile point in an appropriate space where the images of the workspace obstacles are also reported, the space where every possible configuration is represented by the point is called the **configuration space (C)** (**Figure 2b**) of the robot. This associates to each posture of the robot a point in the configuration space. For the planar two arm robot the configuration is q_1 (*joint angle 1*) and q_2 (*joint angle 2*) and the configuration space is all possible values these angles can take. Any sample (*very small area or group of small areas*) taken from this configuration space is called **configuration sample**. It must be noted that configuration spaces here are not Euclidean spaces

but **manifolds[1]** shown in **figure 2c**. However, these manifolds can be treated as Euclidean spaces if a very small area is considered **(lt area approaches zero)**.

In order to characterize paths that represent a solution to the canonical planning problem, those that avoid collisions between the robot and the workspace obstacles. The 'images' of the obstacles are built in the configuration space of the robot shown in **figure 2a and 2b**. it is assumed that the obstacles are closed (*i.e., they contain their boundaries*). Given an obstacles *Oi (i = 1,...,p)* in **W**, its image in configuration space C is called *C-obstacle* and is defined as:

$$COi = \{q \in C: B(q) \cap Oi \mathrel{!}= \emptyset\}$$

In other words, *COi* is the subset of configurations that cause a collision (*including simple contacts*) between the robot and the obstacle *Oi* in the workspace. The union of all is called **C-obstacles** and the compliment is called **free configuration space**, that is, the subset of robot configurations that do not cause collision with the obstacles. A path in configuration space is called free if it is entirely contained in $C_{free.}$ Although C in itself is a connected space, given two arbitrary configurations there exists a path that joins them but the free configuration space C_{free} may not be connected as a consequence of occlusions due to C-obstacles.

It is now possible to give a more compact formulation planning problem. Assume that the initial and final posture of the robot in **W** are mapped to the corresponding configurations in **C,** respectively called start configuration q_{start} and goal configuration q_{goal}. Planning a collision free motion for the robot means then generating a safe path between q_{start} and q_{goal} if they belong to the same connected component of C_{free}, and reporting a failure otherwise.

Manifold[1]*: in mathematics, a generalization and abstraction of the notion of a curved surface; a manifold is a topological space that is modelled closely on Euclidean space locally but may vary widely in global properties*.

Forward kinematics[2]: *refers to the use of the kinematic equations of a robot to compute the position of the end-effector from specified values for the joint parameters*

Collision[3]: *If (q) is the configuration of the robot and B(q) is the volume occupied by Robot, $o_{bstacle}$ is the configuration of the obstacle and B($o_{bstacle}$) is the volume occupied by the obstacle, then for robot not to collide with the obstacle C(B(q)) intersect C(B($o_{bstacle}$)) should be null.*

After introduction to the basic definitions and terminologies, it is time to discuss RRT.

These are the probabilistic algorithms used for path planning in high dimensional (*higher degrees of freedom*) configuration spaces **(*C*)**. The configuration space dimensions depend on the degrees of freedom of the agent, the higher the degrees of freedom for the system, the more complex will be the configuration space for example the robot with six joints (*six degrees of freedom*) is more complex to control then robot with two joints (*two degrees of freedom*).

Using the normal traditional edge constrained algorithms in this space will be computationally intensive, resulting in a high trade-off in the computational time. To work in such high dimensional configuration space, mainly RRT algorithms are applied. These are the probabilistic algorithms based on pure sampling (*where the configuration samples are randomly extracted from the configuration space*).

The idea of the method is simple and is based on the simple recursive iteration of extracting the configuration sample (*set of configurations*) and then using these extracted configurations

(*joint parameters*) in **"forward kinematics[2]"** to determine whether these configurations cause a **collision[3]**. If these configurations are **collision-free** then it is added to the **list** of the collision free configuration or else ignored. This recursive iteration results in the list (*cloud)* of the collision-free configurations C_{free}. Connecting these collision-free configurations results in a network of paths called the **roadmap**. Any heuristic-based algorithm can be used then used to navigate on the generated paths (*roadmap*) from the start node to the goal node.

The two main algorithms under this category are:

i. Probabilistic Roadmap Method:

The basic iteration of the PRM (Probabilistic Roadmap) method begins by generating a random sample $\mathbf{q_{rand}}$ of the configuration space using a uniform probability distribution in **C**. Then, q_{rand} is tested for collision, by using forward kinematics to compute the corresponding posture of the robot and invoking an algorithm that can detect collisions (*including contacts*) between the robot and the obstacles. If q_{rand} does not cause collisions, it is added to the roadmap and connected (*if possible*) through free local paths to sufficiently '**near**' configurations already in the roadmap. Usually, '**nearness**' is defined on the basis of the Euclidean distance in C, but it is possible to use different distance notions. The generation of a free local path between q_{rand} and a near configuration $\mathbf{q_{near}}$ is delegated to a procedure known as **local planner**. A common choice is to throw a rectilinear path in C between q_{rand} and q_{near} and test it for **collision[3]**, for example by sampling the segment with sufficient resolution and checking the single samples for collision. If the local path causes a collision, it is discarded and no direct connection between q_{rand} and q_{near} appears in the roadmap. **Figure 3a** shows an example of PRM and the solution to a particular problem

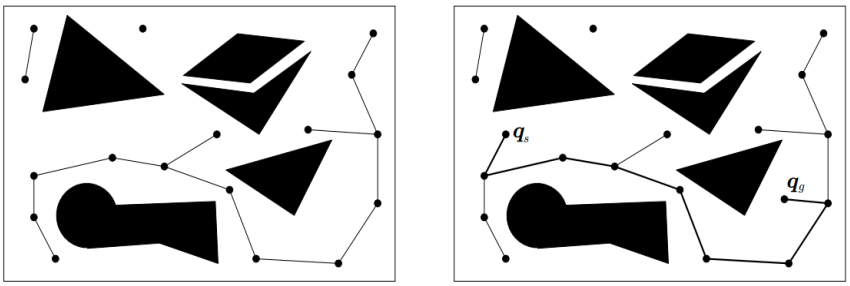


Figure 3a

A PRM in a two-dimensional configuration space (left) and its use for solving a particular planning problem (right)

The PRM incremental generation procedure stops when either a maximum number of iterations has been reached, or the number of connected components in the roadmap becomes smaller than a given threshold. At this point, one verifies whether it is possible to solve the assigned planning problem by connecting q_{start} and q_{goal} to the same connected component of the PRM by free local paths.

Once the process completes it would have generated the network of paths called the **roadmaps**, these roadmaps can connect any two points within the configuration space provided they are on the network of paths. If the start and the goal configuration are not present on the roadmap, then they are selected as new samples and the above-mentioned

algorithm is repeated until they get connected to the roadmap. After that, any graph search algorithm can be used to get the optimal path.

The use of uniform probability distribution for the sample generation has one disadvantage, it hardly selects the sample from the narrow passages. The sample selection is directly proportional to the volume since the volume of the **"narrow passages" (Figure 3b)** is smaller than the rest of the space, therefore, it minimizes the chance of sample to be selected from the narrow passages. If the two regions of the space are joined only by the narrow passage, then there are very high chances that they will remain **disconnected** by the above algorithm.

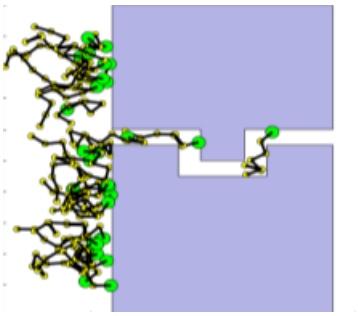


Figure 3b: *Narrow passage and corresponding Roadmap.*

The above-mentioned disadvantage can be overcome by using the selection technique which minimizes the disconnected components. If the start and goal are in the same component then this algorithm is resolution complete. If the two components are disconnected by an obstacle, then it will never add the path between them.

Some important points about this algorithm are:

- It is multi-query algorithm.
- It is fast but computationally intensive.
- It is preferred only when the configuration space is very high (for example in 6D, it finds a solution in some seconds).
- Narrow passages are critical and can be heuristically designed for the non-uniform probability distribution.

The construction step of the algorithm can be shown as:

```
(1)   N ← ∅
(2)   E ← ∅
(3)   loop
(4)      c ← a randomly chosen free
          configuration
(5)      N_c ← a set of candidate neighbors
           of c chosen from N
(6)      N ← N ∪ {c}
(7)      for all n ∈ N_c, in order of
          increasing D(c,n) do
(8)        if ¬same_connected_component(c,n)
            ∧Δ(c,n) then
(9)                E ← E ∪ {(c,n)}
(10)               update R's connected
                    components
```

"Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces Lydia E. Kavralu, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars"

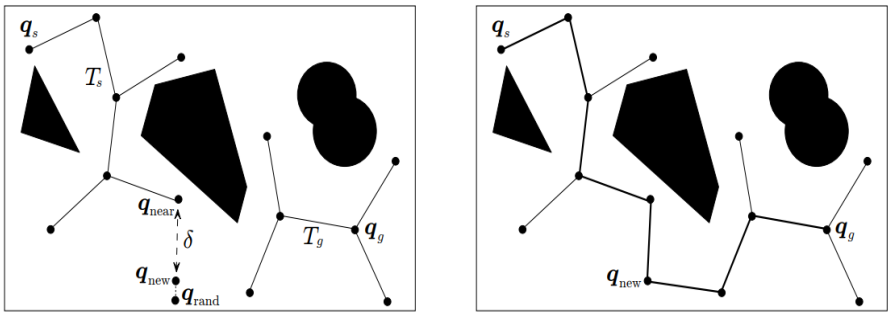
ii. Rapidly Exploring Random Trees:

It is a single-query probabilistic methods aimed at quickly solving a particular instance of a motion planning problem. Unlike **multiple-query planners** such as PRM, this technique does not rely on the generation of a roadmap that represents exhaustively the connectivity of the free configuration space; in fact, it tends to explore only a subset of C_{free} that is relevant for solving the problem at hand. This results in a further reduction of the time needed to compute a solution.

The incremental expansion of an RRT, denoted by **T**, relies on a simple randomized procedure to be repeated at each iteration as shown in **Figure 4**. The first step is the generation of a random configuration q_{rand} according to a uniform probability distribution in **C** (*same as in the PRM method*). Then, the configuration q_{near} in **T** that is closer to q_{rand} is found, and a new candidate configuration $\mathbf{q_{new}}$ is produced on the segment joining q_{near} to q_{rand} at a predefined distance δ from q_{near}. A collision check is then run to verify that both q_{new} and the segment going from q_{near} to q_{new} belong to C_{free}. If this is the case, **T** is **expanded** by incorporating q_{new} and the segment joining it to q_{near}.

The q_{rand} is not added to the tree, so that it is not necessary to check whether it belongs to C_{free}; its only function is to indicate a direction of expansion for **T.** It is worth pointing out that the RRT expansion procedure, although quite simple, results in a very efficient '*exploration*' of C. In fact, it may be shown that the procedure for generating new candidate configuration is intrinsically biased towards those regions of C_{free} that have not been visited yet. Moreover, the probability that a generic configuration of C_{free} is added to the RRT tends to 1 as the execution time tends to infinity, provided that the configuration lies in the same connected component of C_{free} where the RRT is rooted.

This process is recursively iterated till we get the whole space connected or the goal is reached. At this point, we can use any tree search algorithm on the generated tree to get the optimal path from the start node to the goal node. This algorithm has a clear advantage over the previous one, it is biased towards the goal; thus, is much faster compared to the roadmap algorithm. There are many variants of this algorithm that are out of scope for this article.



The bidirectional RRT method in a two-dimensional configuration space left: the randomized mechanism for expanding a tree, right: the extension procedure for connecting the two trees

Figure 4

For a general configuration space ***C***, the algorithm in pseudocode is as follows:

Algorithm BuildRRTss

Input: Initial configuration q_{init}, number of vertices in RRT K, incremental distance Δq
Output: RRT graph G

G.init(q_{init})
for k = 1 to K do
q_{rand} ← RAND_CONF ()
q_{near} ← NEAREST_VERTEX (q_{rand}, G)
q_{new} ← NEW_CONF (q_{near}, q_{rand}, Δq)
G.add_vertex(q_{new})
G.add_edge (q_{near}, q_{new})
return G

- "←" denotes assignment. For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the following value.

In the algorithm above, "RAND_CONF" grabs a random configuration $\boldsymbol{q_{rand}}$ in ***C***. This may be replaced with a function "RAND_FREE_CONF" that uses samples in $\boldsymbol{C_{free}}$, while rejecting those in $\boldsymbol{C_{obs}}$ using some collision detection algorithm. "NEAREST_VERTEX" is a function that runs through all vertices *v* in graph *G*, calculates the distance between $\boldsymbol{q_{rand}}$ and ***v*** using some measurement function thereby returning the nearest vertex. "NEW_CONF" selects a new configuration $\boldsymbol{q_{new}}$ by moving an incremental distance $\boldsymbol{\Delta q}$ from $\boldsymbol{q_{near}}$ in the direction of $\boldsymbol{q_{rand}}$.

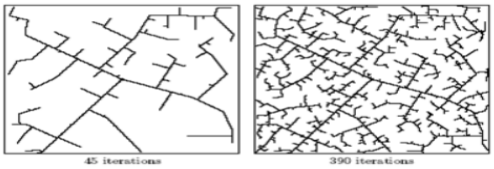


Figure 5: *RRT algorithm at 45^{th} (left) and 390^{Th} (right) iteration.*

2. Grid based A* Any Angle Algorithms for Path Planning

Before diving into the details of the **A*** based algorithms we first need to sketch the analysis to determine how much longer the **shortest grid path** can be than the **actual shortest paths**. We differentiate among the different types of paths like **grid paths** (formed by the line segments whose endpoints are visible neighbors), **vertex paths** (formed by the line segments whose vertices are visible), and **mixed paths** formed by the combination of the above two paths.

The shortest paths are never longer than the shortest vertex path and the shortest vertex paths are never longer than the shortest grid paths.

There are two main steps in sketching this analysis. Firstly, for every line segment of the shortest vertex path, the ratio of the lengths of any shortest grid path with the same endpoints is not affected by the blocked grid cells. This is done by showing that the shortest grid path traverses only the interior of those grid cells that the vertex segment also traverses. Secondly, for all possible endpoints of a vertex line segment, the worst-case ratio of the lengths of any

shortest grid path with the same endpoints is maximized. This analysis provides the upper bound on the ratio of lengths. **Figure 6** depicts both of these analyses.

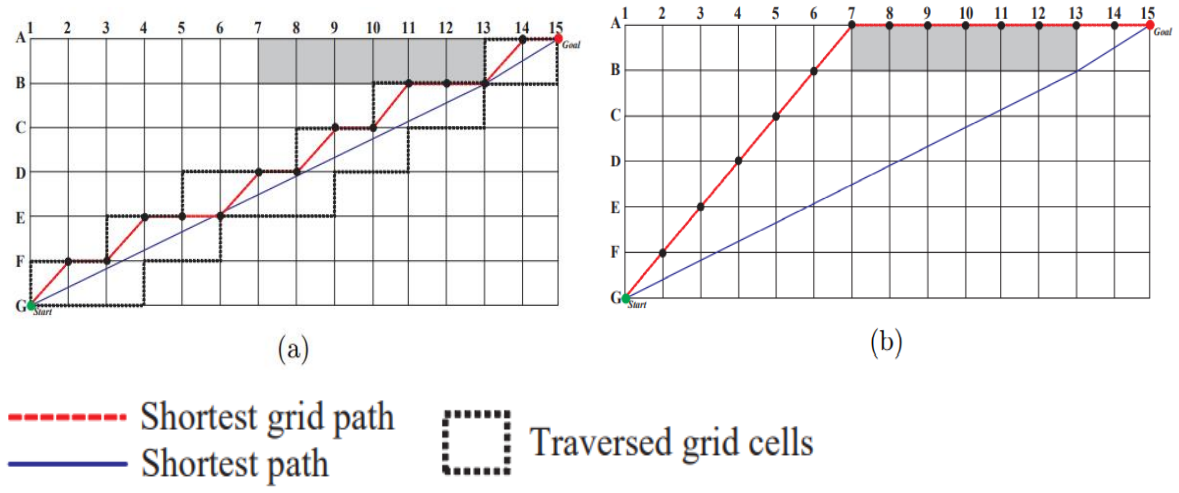


Figure 6

Dimension	Regular Grid	Neighbors	Ratio	In Relation to Shortest Vertex Paths	In Relation to Shortest Paths
2D	triangular grid with vertices at corners	3-neighbor	= 100%	tight	tight
		6-neighbor	≈ 15%	tight	tight
	square grid with vertices at corners	4-neighbor	≈ 41%	tight	tight
		8-neighbor	≈ 8%	asymptotically tight	asymptotically tight
	hexagonal grid with vertices at centers	6-neighbor	≈ 15%	tight	lower bound
		12-neighbor	≈ 4%	asymptotically tight	approximate lower bound
3D	cubic grid with vertices at corners	6-neighbor	≈ 73%	tight	lower bound
		26-neighbor	≈ 13%	asymptotically tight	approximate lower bound

Table 1: Path-Length Analysis of Shortest Grid Paths (Nash 2012)

For the remainder of this article, we will use a **2D 8-neighbor** square grid (**Figure 7, left**) and 3D **26- neighbor** cubic grids (**Figure 7, right**), both with vertices placed at the **corners** of the grid cells. These cases allow us to generalize from 2D to 3D terrain.

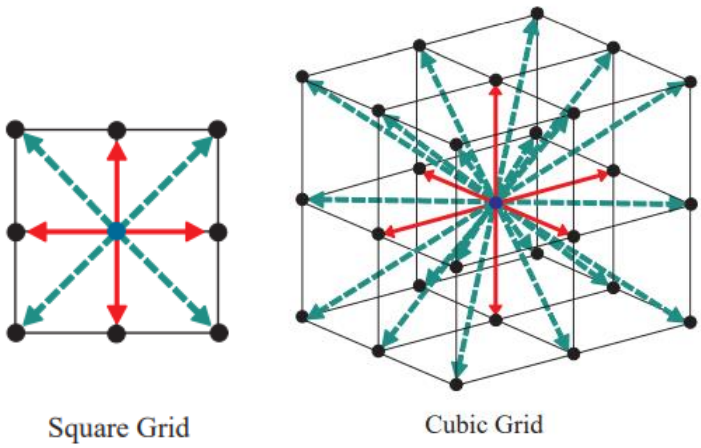


Figure 7

All the path planning algorithms going to be discussed now will be using **A***. Before discussing any angle algorithms, a brief description of **A*** is provided.

A*

The most commonly used algorithm in Artificial Intelligence and Robotics is **A***. It is based on the heuristics **(h(n))**, which the user has to provide before the application of the algorithm. The heuristic for a node **(h(n))** is the estimated distance of a node from the goal node. For the description of the **A***, we assume that all paths are constrained to the edges of the graph made by connecting the neighboring vertices. The **h(n)** value should be **consistent, admissible** and should follow the **triangle inequality**.

It evaluates nodes by combining **g(n)** (the cost to reach the selected node), and **h(n)**, the cost to get from the selected node to the goal:

$$\mathbf{f(n) = g(n) + h(n)}.$$

Since **g(n)** gives the path cost from the start node to node **n**, and **h(n)** is the estimated cost of the cheapest path from **n** to the goal, we have

$$\mathbf{f(n)} = \text{estimated cost of the cheapest solution through n}$$

Thus, to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of **g(n) + h(n)**. It turns out that this strategy is more than just reasonable: provided that the heuristic function **h(n)** satisfies the certain conditions stated above. **A∗** search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that **A∗** uses **g + h** instead of **g**. **Figure 8** and **9**, explain **A*** on two different use cases.

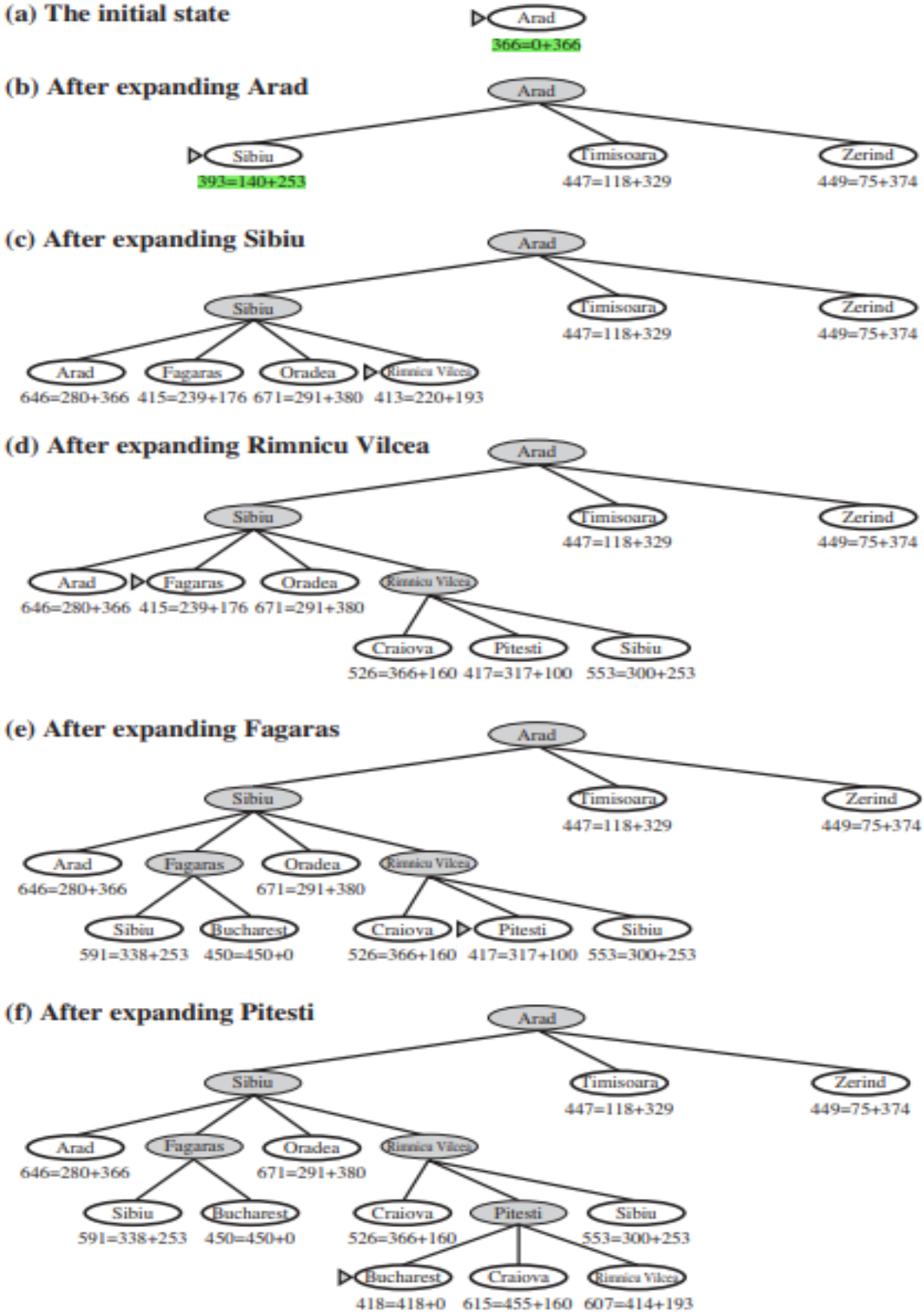

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

Figure 8

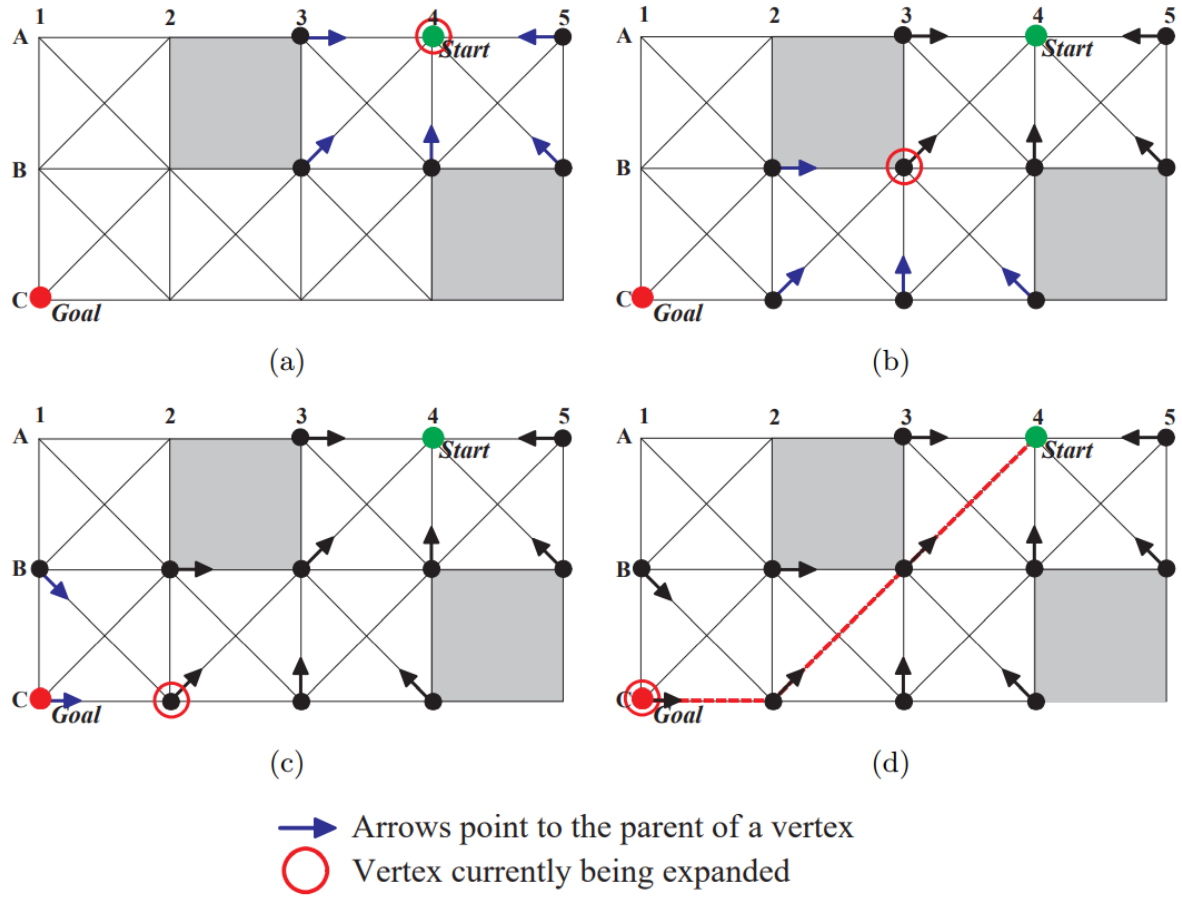


Figure 9

Any Angle path planning algorithms

A* algorithm finds the path in a short interval of time, but the path founded by **A*** is not the shortest one (**optimal one**), on another hand, the path generated by **visibility graphs** is shorter than edge constrained paths, but takes a longer time. Any angle algorithms combine the best of these two worlds. Like the **A*** any angle algorithms share the information along the edges but it doesn't constrain the paths to the edges, rather allows them to have any angle which reduces the unnecessary **head changes**.

There are three important properties to measure the success of Any Angle algorithms over the **A***:

- **Efficiency:** Any Angle algorithms should be faster than **A*** on **visibility graphs** and should find the shorter paths compared to Grid constrained paths.
- **Simplicity:** Any Angle algorithms should be simple to understand, implement, debug and extend
- **Generality:** Any Angle path planning algorithms should be applicable to every graph embedded in **2D** or **3D**, irrespective of the **discretization** technique used.

A very simple Any Angle path planning is **A*** with **post smoothing**, in this algorithm, firstly, the **A*** is being executed on the graph (grid) to generate the shortest grid path and then this

path is being smoothened by some smoothing technique (i.e., to remove the unnecessary heading changes). **A*** with post smoothing typically finds the shorter path, however, this is not guaranteed. It typically leaves the topology of the path unchanged and thus is not guaranteed to find the shortest path. **Figure 10** shows the execution of the **A*** with **post smoothing**.

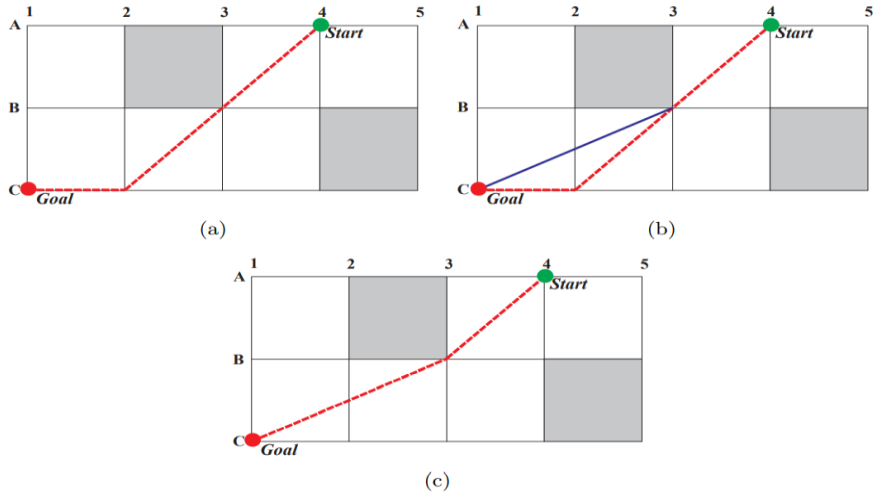


Figure 10

Thus, in short, the **A*** with the post smoothing doesn't have any bias towards the shorter topology, therefore one might want to interleave smoothing with the **A*** search, which adds **bias** to shorter topologies while selection. There exist many ways of interleaving the **A*** search with the smoothing. We discuss three of them in this article, resulting in different Any Angle path planning algorithms, namely **Block A***, **Field D***, and **Accelerated A***.

i. Block A*:

It generalizes **A***, instead of expanding a node of the grid per iteration it expands **the block of size (m × n)** region per iteration. The local distance database **(LDDB)** is created. For every block, the paths are precomputed from every fringe vertex of the block to every other fringe vertex then their path lengths are being stored in **LDDB.** These paths can be the shortest grid paths or any other paths depending on the planning algorithm applied to each block. If the applied algorithm to each block is Any Angle algorithm, then stored paths are **Any Angle paths**. During the path search from the start node to the goal node, **Block A*** uses this lookup table to perform the smoothing before the application of the A*.

A* and Block A* mainly differ in the way they expand, a node or a block respectively. Consider **Figure 11**, where 2 × 2 blocks of unobstructed tiles are expanded. For simplicity, assume a **4-way** vertex grid with a **zero heuristic** is used. The block expansion could be done by following four phases:

a) **Pre-Expansion:** The block's parent is being considered from the south. The vertices which are circled are called **Ingress vertices**, these vertices are from parent block with finite **g-value**. The vertices which are present on the boundary of the block are called **egress vertices**.
b) **LDDB loaded:** In this step the LDDB entry for the block pattern is retrieved, LDDB **(y, x)** is used to return the shortest path between any **ingress y** and the **egress x** for a

given obstacle. As seen in the figure there are three ingress vertices therefore the above function would be called separately for each ingress vertices.

c) **LDDB applied:** In this step the g value of ingress vertices is added to **x** to find the g value of the shortest path from start to every **x** on boundary.

d) **Post-Expansion:** For each egress **x** the new **g-value** is updated to the minimum of the smallest g-value from all paths via this ingress.

Finally, the four neighbor blocks of the expanded block are placed into the **open list** using the minimum updated **s′ (s′. g + s′ .h)** as its **heap value**, where **s′** is the boundary vertices shared by the neighboring blocks and expanded block.

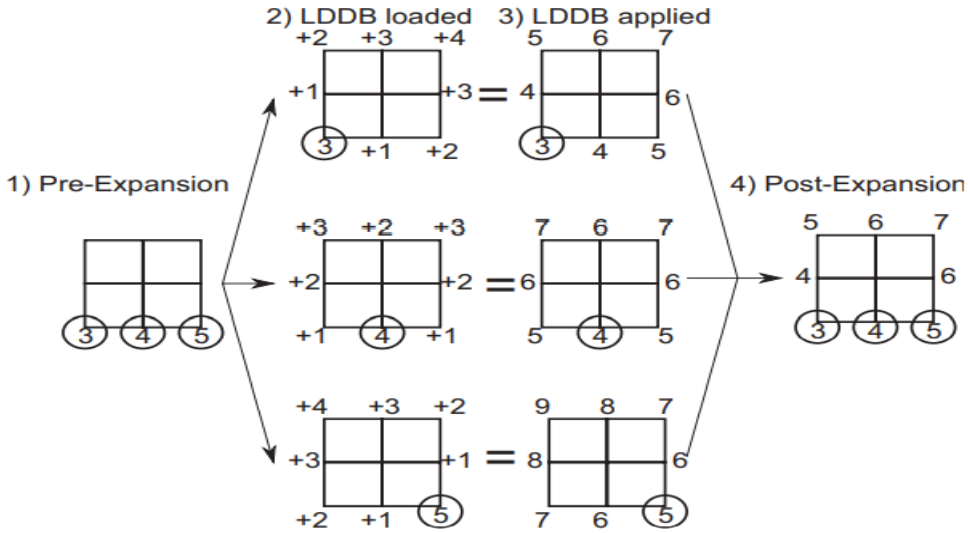


Figure 11

Now to understand the execution trace of **Block A*** check **Figure 12**. The square grid is being partitioned into six blocks of size **(5 × 5).** The LDDB contains the path lengths of each block from start **fringe** vertices **(s)** to end **fringe** vertices**(s′)**. The start vertex is L4 and the goal vertex is D4. Straight line distance is being used as the heuristic. **Block A*** starts by expanding the block K1-K6-P6-P1, this block contains the start node. The **f-value** of the fringe vertices (s) would be calculated by the method of **"block expansion"** as explained above. Then all the neighbors of the selected block will be generated (F1-F6-K6-K1, K6-K11-P11-P6, and F6-F11- K11-K6) and the **f-value** of each block will be calculated. These generated blocks will be stored in the **open list** against their keys, which is the smallest f-value for each block. If the block is already in the list, then the key value is updated only if the generated f-value is lower than the stored key value. This procedure is recursive until the block containing the goal vertex is reached. Finally, the path extraction retrieves the shortest path [L4, L5, H5, H4, J3, J2, F2, D4].

This algorithm can be extended to all types of 2D grids, however, the path found may not be the shortest one, because of unnecessary heading changes in free space especially in fringe vertices. Good thing is that the generated path can be easily smoothened by the post smoothing process. The biggest trade-off of this algorithm is the memory usage by lookup table and determining the size of the block.

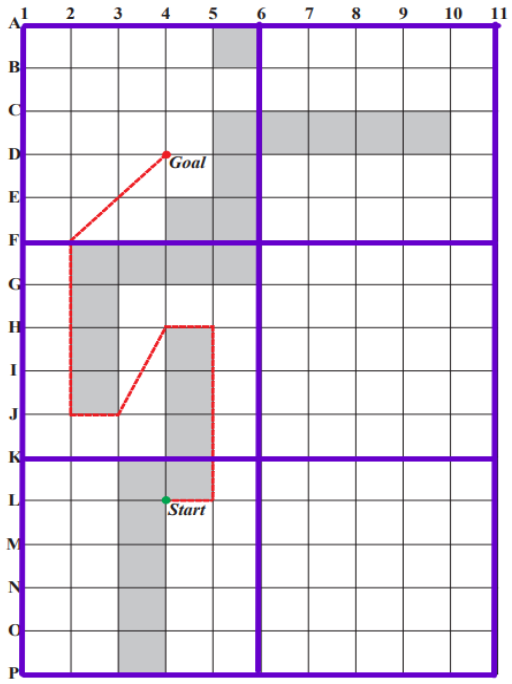


Figure 12

Pseudo code of the algorithm is:

Algorithm 2 Block A*

```
1: PROC: Block A* (LDDB, start, goal)
2: startBlock = init(start)
3: goalBlock = init(goal)
4: length = ∞
5: insert startBlock into OPEN
6: while (OPEN ≠ empty) and
   ((OPEN.top).heapvalue < length) ) do
7:    curBlock = OPEN.pop
8:    Y = set of all curBlock's ingress nodes
9:    if curBlock == goalBlock then
10:      length = min_{y∈Y}( y.g+dist(y, goal), length)
11:   end if
12:   Expand( curBlock, Y )
13: end while
14: if length ≠ ∞ then
15:   Reconstruct solution path
16: else
17:   return Failure
18: end if
```

ii. Field D*:

Field D* uses the interpolation between the **g-values** of the vertices to calculate the **g-value** of the **non-vertex** location. The main difference between the **A*** and **Field D*** is in the calculation of the g-value and updating of the parent of the vertex. When the node **(s′)** is selected for the expansion from the expanded node **(s)**, Field D* considers setting the parent of **(s′)** to any vertex **(v)** or non-vertex **(x)** location that is visible from **(s′)**. The non-vertex location is calculated by interpolating the vertices which are present on the perimeter of **(s′)** and can be visible from **(s′)**. The parent is updated to non-vertex location **(x)** if the path length from start to **(s′)** is shorter via **(x)** than **g(s′)**. The resulted path length will be **g (x′) + c (x′, s′)**, here **c (x′, s′)** is cost of moving from **x** to **s′**.

Figure 13 is showing the **field D*** in operation on a **2D**, 8-neighbour square grid with vertices placed at the corners of a grid. The start vertex is **C1**. The perimeter of **s′ (B4)** is highlighted by the thick border. Any arbitrary non-vertex location **x** is being selected on the perimeter of **s′**. The **g-value** of **x** is calculated by the linear interpolation of two vertices on the perimeter that are closer. Now the g-value of the **x** is different here than the shortest path as depicted in **Figure 13**. As the **g (x′) + c (x′, B4)** is smaller than **g(B4)**, the linear interpolation predicts that there exists the shortest path from the start to the goal via non-vertex location **x.** There exist infinitely many non-vertex locations **x′** on the perimeter, however, the optimization problem can be used to solve the non-vertex locations that minimize **g(x′) + c(x′, B4)** on each of the eight grid edges.

The biggest disadvantage of the field D* is that sometimes it **miscalculates** the non-vertex location which incorporates the unnecessary heading changes in the path thus increasing the path length. The field D* uses only **one step ahead post-processing** smoothing, thus, even if it removes some of the head changings there would still be many head changes left unaltered.

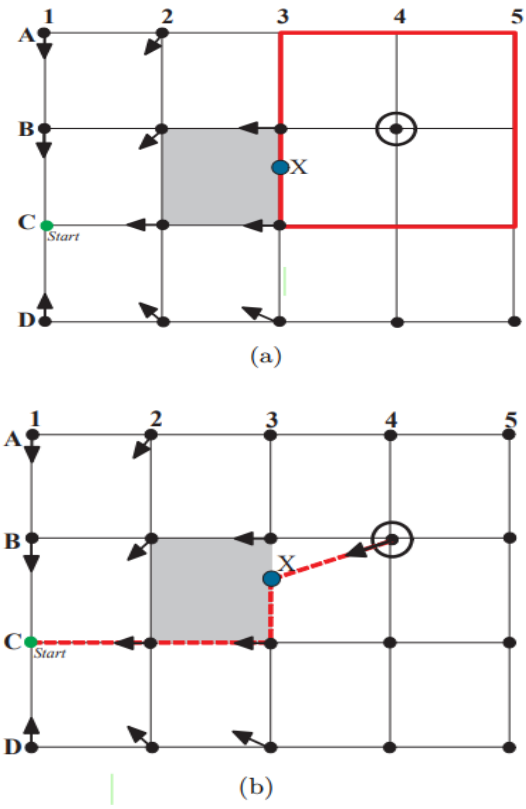


Figure 13

The pseudo code of the algorithm is:

key(s)
01. return $[min(g(s), rhs(s)) + h(s_{start}, s); min(g(s), rhs(s))]$;

UpdateState(s)
02. if s was not visited before, $g(s) = \infty$;
03. if $(s \neq s_{goal})$
04. $rhs(s) = min_{(s', s'') \in connbrs(s)}$ComputeCost$(s, s', s'')$;
05. if $(s \in OPEN)$ remove s from *OPEN*;
06. if $(g(s) \neq rhs(s))$ insert s into *OPEN* with key(s);

ComputeShortestPath()
07. while $(\min_{s \in OPEN}(\text{key}(s)) \dot{<} \text{key}(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start}))$
08. remove state s with the minimum key from *OPEN*;
09. if $(g(s) > rhs(s))$
10. $g(s) = rhs(s)$;
11. for all $s' \in nbrs(s)$ UpdateState(s');
12. else
13. $g(s) = \infty$;
14. for all $s' \in nbrs(s) \cup \{s\}$ UpdateState(s');

Main()
15. $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty$;
16. $rhs(s_{goal}) = 0; OPEN = \emptyset$;
17. insert s_{goal} into *OPEN* with key(s_{goal});
18. forever
19. ComputeShortestPath();
20. Wait for changes in cell traversal costs;
21. for all cells x with new traversal costs
22. for each state s on a corner of x
23. UpdateState(s);

iii. Accelerated A*:

The algorithms discussed till now may have better results (shorter paths) if the world is **schematized** with a greater number of cells, but if the number of cells will be increased it will increase the time to generate the **shortest path**, because, the above-discussed algorithms would have to expand more vertices. Moreover, it could lead to **memory starvation**. In this type of situation, the **accelerated A*** may come handy. To reduce the dimension of the search space **Accelerated A*** uses the specific selection of the successor to be explored. In fact, this selection of the nodes for the expansion of the algorithm accelerates it. In every step, only the limited number **(four)** of nodes are being selected for expansion.

The two main steps which make this algorithm different from its parents **A* and Theta*** are as follows:

Dynamic adaptive expansion: At each step, only four successors will be selected, and the search will be guided only in the direction of one out of these four successors.

Progressive Truncation: in order to obtain the shortest path, truncation is applied to avoid the passage over potentially useless nodes.

Given the **starting vertex (v)** and the **goal vertex(g)**, the algorithm starts from the start vertex and makes the ***"maximum unblocked square"*** around the start node. This unblocked square as the name suggests is the regular shape (square here) of maximum number of unblocked or unobstructed cells around the **centre**, The centre is the selected node (initially starting vertex). The important point about this block is that if the goal node comes inside it, it must be on its **perimeter**. Once the block is made, we consider the successors for the expansions which are close to the selected node (centre node). The term close here means the nodes which are present in the **4 cardinal** directions of the selected node.

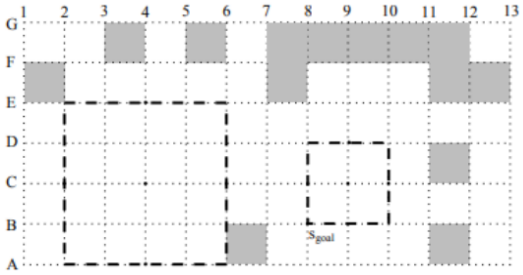


Figure 14

Figure 14 shows the block expansion of two nodes using the **Accelerated A*.** The **maximum unblocked block** for the C4 is the bigger one, which is depicted in the left part of the figure, and the maximum unblocked block for C9 is the smaller one, in which the goal node falls inside it. The **four cardinals** for the C4 as can be seen from Figure 14 are (E4, C6, A4, C2), which are selected as the successors of C4. The paths from the center of the block to these nodes are definitely shorter.

To truncate the **useless** nodes, the **accelerated A*** takes into account the already visited nodes which are present in the **closed list**. The potential parent has been looked at, has to satisfy the two requirements. Firstly, it must be connected to vertex and secondly, its **g value** should be minimal. It is important that this algorithm doesn't traverse all the vertices present in the **closed list**, but only those vertices are traversed which can pass the **ellipse test**. The boundary of the ellipse underlines the nodes which are used to reconstruct the **minimum path** starting from the start vertex to any other vertex via **mid node**. In the construction of such an ellipse, two nodes are placed in two **foci of the ellipse**, knowing their g-value, the two radii can be found (for formulas check the paper). Figure 15 shows the ellipse test.

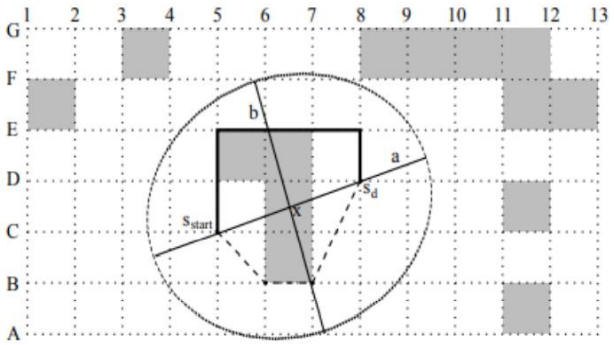


Figure 15

Here is the pseudo code of the algorithm taken from original paper:

```
{1}  Search(s_start, s_goal)
{2}      g(s_start) ← 0;
{3}      h(s_start) ← c(s_start, s_goal);
{4}      parent(s_start) ← false;
{5}      OPEN ← {s_start};
{6}      CLOSED ← ∅;
{7}      while OPEN ≠ ∅ do
{8}          s_c ← RemoveTheBest(OPEN);
{9}          if s_c = s_goal then return s_c;
{10}         Insert(s_c, CLOSED);
{11}         foreach s_d ∈ 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) ← g(s_c) + c(s_c, s_d);
{15}             h(s_d) ← c(s_d, s_goal);
{16}             parent(s_d) ← s_c;
{17}             ProcessNode(s_d);
{18}         end
{19}     end
{20}     return false;
{21} end
{39} Candidates(s_c)
{40}     sq ← DetectMaxSquare(s_c);
{41}     return UsableSideCenters(sq);
{42} end
{43} ProcessNode(s_d)
{44}     foreach s_n ∈ 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) ← g(s_n) + c(s_n, s_d);
{48}                 parent(s_d) ← s_n;
{49}             end
{50}         end
{51}     end
{52}     InsertOrReplaceIfBetter(s_d, OPEN);
{53} end
```

3. References:

I. Any-Angle Path Planning *Alex Nash Northrop Grumman Integrated Systems Carson, California 90746, USA alex.nash@ngc.com Sven Koenig Computer Science Department University of Southern California Los Angeles, California 90089-0781, USA skoenig@usc.edu*

II. Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces *Lydia E. Kavralu, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars*

III. Theoretical Overview for Geometric A* path-planning based approaches *Matteo Emanuele*

IV. Abstract: Block A* and Any-Angle Path-Planning *Peter Yap and Neil Burch and Robert C. Holte and Jonathan Schaeffer*

V. Automated Planning and Acting *Malik Ghallab LAAS-CNRS, University of Toulouse, France Dana Nau University of Maryland, USA Paolo Traverso FBK ICT IRST, Trento, Italy*

VI. A Dissertation ANY-ANGLE PATH PLANNING *by Alex Nash*

VII. Field D*: An Interpolation-based Path Planner and Replanner Dave Ferguson and Anthony Stentz