JPS Algorithm Adaptation and Optimization to Three-dimensional Space

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos
Ohjelmistotekniikka
Diplomityö
Pertti Ranttila
Kesäkuu 2019

TURUN YLIOPISTO

Tulevaisuuden teknologioiden laitos

RANTTILA, P: JPS-algoritmin soveltaminen ja suorituskyvyn optimointi kolmiulotteisessa avaruudessa

Diplomityö, 82 s.

Tämän diplomityön tarkoituksena on tutkia JPS-polunhakualgoritmin (JPS) periaatteita ja mahdollisuutta soveltaa niitä kolmiulotteiseen ympäristöön. JPS perustuu osin A*-algoritmiin mutta on suorituskyvyltään merkittävästi parempi kuin A*-algoritmi. Tällä hetkellä ei ole tiedossa olevaa julkaistua kolmiulotteista polunhakualgoritmia, joka käyttäisi samoja periaatteita kuin JPS. Työn tarkoituksena on selvittää mitä muutoksia JPS-algoritmi vaatii, jotta sitä voidaan soveltaa kolmiulotteisessa ympäristössä ja saavuttaa mahdollisesti parantunut suorituskyky A* algoritmiin verrattuna. Työssä on vertailtu myös luotujen algoritmien suorituskykyä kolmiulotteisissa ja kerroksisessa ympäristössä (rakennukset) jolla on merkitystä videopeleissä sekä todellisessa maailmassa. Työn käytännön toteutus on tehty käyttäen *Unity3D*-pelimoottoria sekä C#-ohjelmointikieltä. Tutkimuskäyttöä varten algoritmit toteutettiin samaan kolmiulotteiseen testiympäristöön. Järjestely mahdollisti saman testidatan käyttämisen eri algoritmeille ja vertailukelpoisten testitulosten saamisen. Tutkimuksen tuloksena saatiin myös täsmälliset periaatteet, miten käyttää JPS-algoritmia kolmiulotteisessa ympäristössä. Lisäksi työ tuotti uusia ideoita, kolmiulotteisen JPS-algoritmin suorituskyvyn parantamiseksi. Työ sisältää myös suorituskykymittaukset JPS-algoritmin ja sen optimoitujen varianttien välillä erilaisissa testiympäristöissä.

UNIVERSITY OF TURKU

Department of Future Technologies

RANTTILA, P: JPS Algorithm Adaptation and Optimization to Three-dimensional Space

Master Thesis (tech.), 82 p.

The aim of this thesis is to research the principles of the Jump Point Search (JPS) pathfinding algorithm and study the possibilities of adapting JPS to three-dimensional environment. JPS is partly based on A* algorithm but its performance is significantly better than the original A* algorithm. At the moment, there is no known 3D pathfinding algorithm published which uses the same principles as JPS. The motivation of this work is to find out what changes will be needed on the algorithm so that it will work in 3D and improve performance of the 3D version. Special target is performance comparison to the original A* algorithm and 3D JPS algorithm on inside a layered 3D space (building). Pathfinding inside building is common pathfinding problem in a video games as well it has importance also in real life. The algorithms were implemented by using Unity 3D game engine with C# language. For the purpose of research, A* and JPS algorithms were tested in the same 3D test environment. This arrangement made it possible to run the algorithms with the same test data to get a meaningful performance comparison between the algorithms. As a result, this gives also the exact principles how to adapt JPS Algorithm on a 3D environment. Moreover, it provides a novel idea and practical implementation on how to optimize the JPS 3D algorithm to get an improved performance. The presented results also include performance measurements for comparing JPS 3D method and its optimized variant in different environments.

Keywords: JPS, A*, pathfinding, optimization, graph theory

TABLE OF CONTENT

1. INTRODUCTION

Pathfinding refers to methods of finding a path between two given points at the space. The purpose of finding a path is usually related to finding the *shortest path*, which stands for the physical attribute but can also refer to other scalar to be minimized. Finding the shortest path is interesting because of several aspects: In many cases, the shortest path is also the fastest path, it might be also the most energy efficient path. In a modern technological world, pathfinding algorithms are important in many different areas. Household applications like autonomous vacuums or lawn mowers use pathfinding algorithms. Navigation systems are well-known pathfinding applications for many people as well as video games which also rely heavily on pathfinding algorithms. Many industrial applications (e.g., logistic applications) need to be fast and efficient, and, therefore, pathfinding can improve their efficiency. Robotics uses pathfinding technology and lately new technology segments like autonomous vehicles need fast and reliable pathfinding. Growing technology trends like IoT (Internet of Things) will make things smarter and very likely some of those things will also need pathfinding. Telecommunication business utilizes pathfinding technology for ensuring the fastest paths for routing data. Electronic industry like microchip design and circuit board design uses pathfinding for optimizing the product designs. Unmanned aerial vehicles, or drones, have a vast potential to change things like packet delivery, search and rescue and surveillance, and pathfinding is essential for all those applications.

Even if the pathfinding problem can be solved with a relatively simple procedure, it is a memory and computation intensive process. Pathfinding problems are challenging also because different applications have different requirements. There are many different algorithms for solving the pathfinding problem, because there is no single algorithm that would always give the best performance on every environment. Different environments like 2D, 3D, real-time, static, dynamic, single agent, multi agent, and small/medium/large maps have also different requirements for the design of the pathfinding algorithm.

Pathfinding algorithms can be categorized by the quality of the solution they can provide, which can be optimal or suboptimal paths. Depending on the application, sometimes the absolutely shortest path is a priority even if it usually takes a longer time to calculate than a suboptimal path. There are also applications where finding the shortest path is not critical. Finding a suboptimal path is generally a faster process and uses less computational resources. A suboptimal path can also be a better option when the search of an optimal path is practically too heavy and time-consuming process. Applications like video games, where fast response time is very important for the user experience, suboptimal paths can be the best option [1]. Generally speaking, pathfinding is only one part of a process which also includes some action after the path has been solved. The total time of the whole process might be a criterion for choosing a pathfinding algorithm.

One of the most widely used pathfinding algorithm is the A* algorithm [2], which was developed in 1968 in Stanford Research Institute. A* algorithm can be applied in both 2D and 3D environments for finding an optimal path. A* algorithm uses a *graph* for solving the pathfinding problem like earlier developed Dijkstra's algorithm [3]. The major difference is that A* algorithm uses a heuristic value to find the most likely direction of the path, which makes A* algorithm generally more efficient. In practice, A* algorithm works reasonably well on small 2D environments but with larger maps its performance slows down significantly. The performance in a 3D environment slows down even faster as the search space is increasing cubically. Finding an optimal path in 3D *real-time* is a computationally challenging problem. The purpose of this thesis is to research/implement a 3D pathfinding algorithm which is possibly faster than A* algorithm and which can calculate the optimal path online without preprocessing.

2. CHALLENGES ON PATHFINDING

In order to identify factors why the pathfinding process requires a considerable amount of computational and memory resources, one must analyze the pathfinding problem and its essence in a general level.

The most principal element of the pathfinding problem is the environment, which is the area or space where the pathfinding process is conducted. The environment is a digital model usually representing a physical world. The model represents all entities which have some effect on the pathfinding process, and the accuracy of the model varies depending on the requirements for the solution. Typically, the model holds a considerable amount of information about the locations of traversable/non-traversable areas, and distances between locations. The amount of information becomes large as it is relative to the physical size of the environment and selected granularity. The larger the space, the more information it typically holds. Adequate granularity of the model is crucial for providing an accurate and realistic representation of the environment and allowing to find a natural and optimal path.

The pathfinding process is conducted between two locations on the environment, which are generally denoted with *start* and *goal*, representing arbitrary traversable locations of the space. Obstacles represent non-traversable arbitrary locations and forms arbitrary patterns. Even though obstacles represent physical objects like walls and forming larger entities like rooms, from the perspective of pathfinding algorithm, structures are arbitrated. This arbitrariness of location is significant from perspectives of pathfinding algorithm; thus, pathfinding algorithm cannot use any kind of prediction algorithm of what size and shapes obstacles might form.

Obstacles are also independent from the *start* and *goal* locations. As all these entities like *start*, *goal* and obstacles, can vary randomly, their potential combinations become considerably large, and, therefore, a typical pathfinding problem is large and unique. Because the problem is unique, also the solution is unique, which means that pre-calculating all possible optimal paths is generally not a feasible solution in a large scale.

Some pathfinding algorithms use hierarchical methods for addressing this issue [4]. For example, the environment can be divided into smaller sections and the algorithm then pre-calculates the paths inside those areas between a limited number of waypoints. Usually, the waypoints represent exits between the areas. However, this approach cannot guarantee an optimal path as it represents only an approximation of the optimal path [5].

It is characteristic for a digital model of the environment that each separate location holds distance information only to its nearest neighbors, which have their own nearest neighbors, and so on. Together that information is minimal but enough to define the distance between any arbitrary point of locations on the space. The path is chosen by traveling from neighbor to neighbor by using some qualification for selecting the neighbor. In the case of finding the shortest path, the qualification rule is minimizing the total length of the path. The total path length between the start and the goal can generally have a large amount of variations. The amount of variations depends on the number of nearest neighbors (connectivity), how many of those are obstacles and how the neighbors are chosen (algorithm). Clearly, a larger space and, therefore, a longer path length also adds more variations.

The problem of selecting the nearest neighbor is characteristic of the pathfinding process. Solving an optimal path requires finding such a variation of travelled locations that the sum of interlocation distances is minimized (i.e., it represents the shortest path). However, there is not any clear indicator for the length of the shortest path. Even if we have known the optimal path, nothing on the path itself, unless it is a straight line, indicates that the path is really optimal. Algorithm cannot reveal the optimality of a curved path without considering the environment. To validate that the path is optimal one needs information that there is no other alternative path which could be shorter. To get that information computational resources are needed for processing alternatives paths. Each of those paths has its own alternative branches to consider. In a large complex environment, there are numerous potential paths and the pathfinding turns into a challenging combinatorial problem.

In conclusion, the pathfinding is an iterative process producing sub-solutions, where an everything-at-once approach is not feasible. The entire solution is a sum of individual

solutions, these solutions represent individual algorithmic decisions to choose the next direction to travel. These decisions are made often based on heuristic functions which have only a limited accuracy. To find out the length of the entire path requires evaluating all the nodes along the path. If the path is not sub-optimal, it is impossible to alter the individual choice along the path without recalculating/rerouting the whole path.

3. JUMP POINT SEARCH ALGORITHM

Jump Point Search (JPS) pathfinding algorithm was introduced in 2011 by Daniel Harabor and Alban Grastien [6]. JPS algorithm provides the same optimal path as A* algorithm but it is significantly faster than A, and it also works with about the same memory space requirement as A*. JPS algorithm is based on A* but instead of using a *greedy search* and searching a large number of nodes, it can selectively rule out many nodes that would not lead to any better path than already known (see Figure 1). The developers of JPS refer to this system as *symmetry breaking*, and it is the main reason why JPS can outperform the regular A* algorithm. In many test cases, JPS can gain speed which is about 10 times faster compared with the traditional A* algorithm.

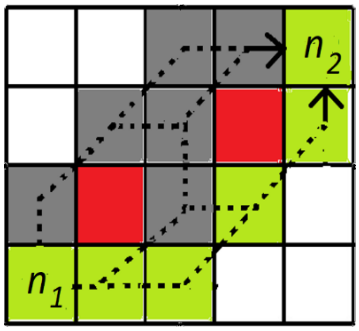

Figure 1. The green nodes represent optimal path between n_1 and n_2. The red nodes represent obstacles. Gray nodes represent possible alternative optimal paths between n_1 and n_2.

With larger search environments, A* has a major challenge when the so-called *openlist* increases. *Openlist* is an essential data structure for storing the nodes which need to be visited. Each node has two values: *gCost* and *hCost*, where *gCost* represents a value from the start node to a node on the path from *start* to *goal* and *hCost* represents the estimated distance to the goal node. If a new and shorter path to the node is found, A* updates the *gCost* values of the node. The sum of the *gCost* and *hCost* is *fCost*, and the most promising node is the node with the lowest *fCost*. Updating and searching these values is computationally expensive, especially on a large scale. To ensure the best possible performance, an efficient data structure for *openlist* is very important. For this reason,

openlist is usually implemented as a *priority queue* which can be supported by a *min heap* [7]. With a larger search environment, the length of *openlist* increases drastically and the slowdown of pathfinding is then inevitable. This issue is a fundamental problem of A* algorithm as long the optimal path is a requirement.

A* algorithm uses a great amount of computational resources for each node. Especially updating the node information is often a recurrent process when the node can be reached via multiple paths. Much unnecessary calculations are carried out, if multiple paths provide an equal or longer path length than the currently known. The recurrent calculation problem comes very clear when the shortest paths are symmetric. Another issue with A* is that it does not rule out nodes which are clearly not even leading to any kind of path at all, such as obvious dead ends or nodes which are way out from the natural shortest path.

JPS algorithm uses a more sophisticated way to handle the nodes. Compared to A* where every node is an equally suitable candidate for the potential path, JPS makes a difference between the nodes depending on which direction the node is approached. It means that the same node might be a possible candidate for a path or totally ruled out depending on the case. This evaluation of possible candidates is done online during the pathfinding process. As a result of this selective evaluation, *openlist* stays significantly smaller compared with the A* algorithm. JPS evaluates the node and adds it to the *openlist* only if the node fulfils the requirement of the so-called *jump point*. The process of searching for the *jump points* creates a much simpler graph for the algorithm to solve the path and, therefore, JPS is less time-consuming than the average A* pathfinding process.

To apply a pathfinding algorithm, it is necessary to create a digital model of the space. This modelling is called *discretization*, and it divides the space with a finite set of coordinates/nodes. There are several different possibilities to discretize an environment. In 3D space, dividing the space on equalized cubic sections gives a uniform grid, which is natural and algorithmically straightforward to handle. In this case, every node of the grid has the same connectivity which gives the same number of adjacent neighbors, excluding the nodes on the outer bounds. The division of the space means that the distance from any node to its nearest neighbors is always exactly predefined. The result of the discretization is a finite set of coordinates which represents all the possible locations in

the space. To have a meaningful pathfinding problem it is assumed that this space is not empty, and it includes also non-passable areas.

Many pathfinding algorithms like Dijkstra, A* and JPS use graphs for modelling the search space. A *graph* is an ordered pair of $G = (V, E)$, where the vertices (nodes) V represent locations and the edges E represent the connections between those locations. A graph presentation does not include information about the actual coordinates of the vertices in the space and it uses only weight values between the vertices for calculating the shortest path.

In a 2D/3D environment, the path refers to a set of nodes/vertices which are connected with line-of-sight connections. JPS and A * algorithms calculate the path always through the nearest neighbors. This arrangement limits the connectivity of a node when only some predefined directions of the path are allowed from the node. Typically, each node has a connectivity to its adjacent nodes/nearest neighbors. Connectivity is negotiable, and it has an effect on the performance and smoothness of the path. A path is polyline, so the optimal path generated with JPS algorithm is not always physically the shortest due the constraints of connectivity. A pathfinding algorithm which allows a line-of-sight connection between any node is called *any angle pathfinding* algorithm [8]. It can provide true shortest path but for doing this is computationally demanding especially in a three-dimensional space.

3.1. NOTATIONS AND PRESUMPTIONS

The original JPS algorithm [6] is a set of formal equations and rules on how the nodes are evaluated during the pathfinding process. Because that description of JPS algorithm includes some domain specific terminology, this thesis follows the terminology of the original paper whenever it is applicable. Later, when the algorithm is modified and expanded to a 3D space, additional terms are introduced and explained. For simplifying the environment and ensuring the correct functionality of the algorithm, some preconditions are set. In this thesis, assumptions are made that the pathfinding *agent* is point-like and, therefore, two-dimensional corner cuttings are allowed for the path. These

preconditions are not a built-in constrain on JPS algorithm and are used only for clarifying explanation of the algorithm.

An environment is a two-dimensional uniform grid with a maximum of eight neighbors around each node, and each node is either traversable or non-traversable. The eight adjacent neighbors represent the allowed moving directions. Generally, the directions are denoted with $\overrightarrow{d}$. If the direction is diagonal (45-degree angle between the x-axis), two additional directions $\overrightarrow{d_1}$ and $\overrightarrow{d_2}$ are denoted. Those $\overrightarrow{d_1}$ and $\overrightarrow{d_2}$ are pointing in a 45-degree angle from the diagonal line $\overrightarrow{d}$. It is noticeable that in practice, $\overrightarrow{d_1}$ and $\overrightarrow{d_2}$ are straight, vertical and horizontal directions. The magnitude of diagonal $\overrightarrow{d}$ is $\sqrt{2}$ and the magnitude of $\overrightarrow{d_1}$ and $\overrightarrow{d_2}$ is 1. A path is denoted by $\pi = (n_0,...,n_k)$, where *n* is a node. It is obvious that the path must represent a set of nodes that are in a particular order. That order of nodes is an essential property during the whole pathfinding process as all path length comparisons require it. For representing the order of nodes, the parent relation (predecessor on the path) of a node is denoted by *p(n)*.

Function *len()* is used for measuring the length of a path. Notation *x* is used for defining a path that excludes node *x* on the path. The term: *jump point* is an essential concept on JPS algorithm, and the formal definition of *jump point* will be introduced later. Generally, *jump point* could be described as a turning point where the path might change its direction.

3.2. OBSTACLES AND ENVIRONMENT

Ideally, the path is always a straight line from *start* to *goal*, but the environment is typically occupied with some obstacles. Those are often modeling physical obstacles, which remain on static locations. From the perspective of *graph traversal* algorithm, obstacles occupy space, which reduces the maximum graph size that the pathfinding algorithm produces for the *graph traversal* algorithm. The smaller graph means also less calculation for the *graph traversal* algorithm. On other hand, obstacles prevent path from traveling directly adding the length of the optimal path which commonly requires more computations than a shorter path. From the perspective of a pathfinding algorithm, the significance of the obstacles varies, depending how those affect on the pathfinding

process. Obstacles preventing a path from travelling an ideal straight line from *start* to *goal* set absolute physical constrains on what is the shortest path length. It is worth noting that the same environment with fixed *start* and *goal* might provide multiple optimal paths. These paths can be travelled independently, and the paths are possibly constrained with a different set of obstacles.

Typically, there are also some obstacles, which are not along the shortest path and physically constraining it. Depending on the pathfinding algorithm and the problem instance, some of those obstacles might be necessary to be consider during the pathfinding process, as the potentially optimal path might travel via those obstacles. Therefore, these obstacles generally increase the complexity of the environment and cause additional computational load without providing an option for the shortest path. Different search algorithms and strategies can have an impact how the search is expanding. Heuristic functions are typical used for guiding the search of a path, so that the unnecessary computational load would be minimum. The rest of the obstacles, if any, are located so that the pathfinding process does not need to consider those at all. From the perspective of the pathfinding algorithm, they are neutral objects, which does not require computational recourses.

To demonstrate how to form an optimal path one can use a simplified example. Let us assume that there is a simple two-dimensional environment with *start* and *goal*. Between these points is one obstacle, preventing path to travel straight. When the optimal path changes its direction because of the obstacle, it travels immediately adjacent to that obstacle. It is also obvious, that there is no purpose to evaluate any path straight toward the obstacle if the obstacle can be bypassed. That is obvious, as the potential shortest path always flows straight trajectories on either side of the obstacles.

Those principles are intuitively clear and simple to understand from the human perspective. To evolve this idea further, one can think how humans are generally doing everyday pathfinding. For example, to find some particular location inside an unfamiliar building, it is typical look for exits like doors and hallways. If the destination is not in sight, it is just necessary to go further and look around the next corner. This process can be repeated until the target has been found. During this search process it is no need to pay

much attention to the non-traversable areas like walls or obvious dead-ends. This is possible because human brains are good at recognizing possible and non-possible paths by pattern recognition. Moreover, if there is no potential path, also estimating the distance information is nonrelevant. By using vision, it is possible to choose the prominent paths and avoid dead-ends like rooms without exits. Even though this example is not about finding the shortest path, it demonstrates that observing visually and analyzing the geometry of the environment before action (moving) is an effective method for the humans.

3.3. ALGORITHMIC CHALLENGE

The challenge in the design of a pathfinding algorithm is how to develop an "intelligent" algorithm to recognize certain patterns to avoid unnecessary searching (computation). In addition, the capability of avoiding obstacles should be computationally efficient. Some algorithms like A* use only simple arithmetic for a evaluating the environment and, therefore, they cannot consider other property of the environment than the node and its distance to the nearest neighbors. It is a straightforward method but computationally expensive. This approach gives only a limited chance to utilize the other properties of the environment like geometry and direction to look up. Even though geometry and direction values cannot be stored on the graph, those are valuable information while building the graph.

Depending on the look-up direction, any obstacle in the environment might become significant or insignificant from the perspective of the shortest path. However, it is impossible to accurately declare in advance which paths and directions on the space are must be evaluated. This is a characteristic of every pathfinding problem and it cannot be avoided. The nature of a pathfinding process is about addressing the uncertainties until the path is found.

Basic *graph traversal* algorithms like A* compare only distances and thus, they cannot make any difference between actual physical directions in the environment. Therefore, A* algorithm is forced to search all around, and to gather a sufficient number of nodes

for evaluating the optimal path. This issue remains on A* algorithm even when the heuristic value guides search path towards the goal. The result is typically a large set of nodes to evaluate and a considerable amount of unnecessary computations.

Rejecting those nodes which will not lead to meaningful results and decreasing the size of the graph requires a more sophisticated algorithm than A*. Generally, the algorithm should avoid evaluation of obvious dead-ends and straight trajectories towards obstacles, if possible. It is noticeable that avoiding those situations requires a mechanism to recognize certain structures on the environment. Inevitably, it is an additional process, which ideally should be as computationally light as possible.

3.4. OPTIMALITY OF THE PATH

Optimality of the path means that the path is the shortest possible one between two locations. However, it is also possible that there will be an alternative path(s) between the same *start* and *goal* locations which has the same length but a different route. Thus, it can be stated that the optimality of a path means that among the set of all possible paths there is no shorter alternative path(s). This definition also emphasizes that the length of the shortest path is verified by comparing it to all other potential paths.

To observe how the optimal path and any arbitrary part of that path is traveling. Let π_a be an optimal path from p_{start} to p_{goal}, and p_i and p_j be two arbitrary points on π_a. Then a part of π_a travels between points $p_i \; and \; p_j$, which is denoted π_{sub}. If there is any alternative path π_{ij} between p_i and p_j, which is shorter than π_{sub}, then the shortest path from p_{start} to p_{goal} would travel via π_{ij}, instead of π_{sub}, which would lead to contradiction, since it was claimed that π_a is the optimal path. Therefore, it is obvious that any arbitrary part of the optimal path is also optimal. Generally, the entire path is a set of organized optimal sub-paths [9]. This fact gives some interesting possibilities for pathfinding. If there is an existing (efficient) method for searching the minimum set (enough for optimality) of sub-paths between *start* and *goal*, then it is possible to combine those sub-paths to minimize the total length of the path. Consequently, the combined path must then be the optimal path between *start* and *goal*.

Obviously, any path consists of sub-paths which are straight and/or curved. The first challenge is to recognize those sub-paths. An optimal path always travels in a straight line when possible, as it is the shortest path between two points. The length of the straight path can be an arbitrary long without affecting its optimality. This is possible because the straight path explicitly defines its own optimality. This straight path can also be just a sub-path of other longer optimal path. In addition, this kind of straight sub-path is trivial to solve as it is possible to create a relatively simple and efficient algorithm for evaluating the length. Such kind of algorithm requires to evaluate further on a certain direction, as long there is an accessible area ahead. All those traversable nodes along that path are counted to the total length of the sub-path. This eliminates unnecessary length comparisons to an alternative path as the straight path itself is optimal, and obviously there cannot be shorter alternative paths to consider. A straight path is trivial to solve but the real pathfinding problem usually consist of some obstacles along the path.

When the path is constrained by obstacle(s) in the environment, it becomes curved. To preserve optimality, the path is often required to travel immediately adjacent to the obstacle. However, solving an optimal arbitrary long curved path is not a trivial task. It is worth noting that a curved path does not explicitly define its own optimality, unlike a straight path. The length of curved path cannot be simply added and assumed that the path is an optimal. Proving optimality requires considering geometry of the environment and evaluating other potentially shorter paths. Additionally, when the path is getting longer, the amount of prominent sub-solutions increases rapidly, and the pathfinding problem becomes computationally heavier to solve.

On the contrary, a curved optimal path consists of a set of optimal sub-paths as the theorem about the optimality proves [9]. To utilize the idea of dividing path to optimal sub-paths requires identifying those shortest sections. In the discretized model, any path consists of finite set of sub-paths, additionally minimum length of any sub-path is exactly defined by the connectivity to its neighbors. As the shortest possible path in a discretized model is a straight path between two adjacent nodes, the shortest curved path can be defined by three separate nodes. That sub-path can be thought to consist of two optimal straight sub-paths which have one common pivot point. Those optimal paths are the start

to pivot point and from the pivot point to the end point. This also follows the idea that any optimal path is always combination of optimal sub-paths.

The idea of the sub-paths is applicable in a uniform discretized environment, as the entire world is divided equally sized and shaped sections (i.e., grids). Therefore, it contains a finite amount of options how the path can change its direction. This makes possible to define a limited set of predefined shortest sub-paths where the length and shape are explicitly and accurately defined.

3.5. CORE IDEA OF JPS

JPS algorithm utilizes the idea of finding a set of *jump points*. *Jump points* are evaluated sequentially based on the search of the previous *jump points*. It is noticeable that between every consecutive *jump point*, there is also an optimal sub-path. Moreover, the maximum distance between consecutive *jump points* is not constrained to the nearest neighbors. This is the most significant difference compared to A* algorithm which evaluates only two adjacent neighbors at a time. JPS algorithm optimizes the distance and searches a direction from the parent node to the neighbor node and its neighbors and so on. This process is repeated until the algorithm finds a new *jump point* or returns a null.

To emphasize the differences between A* and JPS algorithm, one can think a typical situation during a pathfinding process when two nodes are adjacent. Optimizing a path between those locations is non-relevant as the only and shortest path is a straight line. If the path is traveling via three nodes, there are possibly more than one path to consider. With the three sequentially adjacent nodes one can generate several different patterns and path lengths for the possible path. The path might be a straight or curved line between those three nodes. Optimality of those paths depends on the on environment and the obstacles. It is also clear that even if it is possible to define a large number of optimal sub-paths on a space, relevance of those sub-path depends on the current pathfinding problem. As each individual pathfinding problem has only a finite set of possible solutions (optimal paths), the set of the optimal sub-paths is also limited. To reduce

number of the sub-paths to be evaluate, JPS uses a direction parameter to rule out non-relevant options.

The arrangement when several nodes are included on the path length comparison, makes possible to conduct a relevant path optimization. JPS algorithm uses optimizing rules which compare the path length based on the geometry of the environment. This is a fundamental difference between A* and JPS algorithms. The core idea is to recognize a pattern of blocked/non-blocked nodes around the evaluated node. In some specific cases, the pattern around the evaluated node is such that it must be considered as a *jump point*. Even though the term *jump point* refers to a location, the location of a *jump point* represents also a straight and optimal sub-path from its parent node. Another significant property for the *jump points* is that they represent locations where the optimal path potentially changes its direction. This feature of JPS algorithm can be seen equivalent to how humans can recognize a corner on the wall when the hallway makes a 90-degree turn. In that case the discontinuity on the wall, indicates that moving exactly on that corner enables observing further for a possible path.

Using a 2D uniform discretization each node has only a nine *nearest neighbors*. Thus, a path from *p(x)* via *x* to its *nearest neighbors* can form only a small and finite set of different sub-paths. Any other path regardless the length, can be a presented as combinations of those sub-paths. Recognizing those sub-paths requires relatively lightweight computation. It can be done fast with simple binary operations

Searching for the *jump points*/sub-path is obviously a recursive process; because it iterates a set of the rules until the path has been found. Each found *jump point* generates information about the relative entering direction to the node and a new search direction(s) from the node, which is an essential property so that the process can preserve its optimality. The search process begins from the *start* node and all further nodes/directions are evaluated based on the previous evaluations. The entire pathfinding process will end when the *goal* node is reached. The outcome of the algorithm forms a tree-structure. The root of the tree is the *start* node where the tree begins branching. When the final branch is formed on the tree, that branch represents an optimal path from the *start* (root) to *goal* node. Other branches represent paths which did not lead to the optimal path. Even though

there might exist branches which could lead to equally good and optimal paths, those path candidates were not fully evaluated as the optimal path has already been found. This feature to minimize branching is so-called symmetry breaking, which addresses a frequently occurring problem of A* algorithm, when several equally optimal paths are should be evaluated sequentially. It is obvious that every branch requires extra computations. Therefore, each branch that did not lead to the *goal* node, represents wasted computational recourses. Generally, the less there are side branches (nodes) and the shorter they are, the more efficient has been the pathfinding process. It is noticeable that the tendency of branching is not solely related on the efficiency of algorithm. The topology of the environment has impact on branching especially in a complex environment it is inevitable that branching increases as the search process has more prominent paths to evaluate. In summary, the core idea of JPS is fundamentally different from A* algorithm. JPS algorithm utilizes the geometry of the environment and builds a smaller *graph* than the A* algorithm.

3.6. JUMP POINT

For defining any arbitrary path on two/three-dimensional space requires certain minimum information. Obviously, a *start*-point, *goal*-point and a sequence of the points are needed in the order where the path changes its direction. Evaluating any other intermediate points along the path is irrelevant, as it cannot provide any additional information for the path. The challenge is, how to efficiently and accurately discover those turning points between *start* and *goal*.

A* algorithm uses pure arithmetic for resolving the path, and it can be compared to brute force methods, as it evaluates each node in the same way while the search expands. This process generates significantly more nodes than what is the minimum set of nodes to define path with turning points. A more sophisticated way is to determine, if each evaluated node fulfills the minimum requirement of turning point and then add it to the graph. Otherwise, it is safe to reject the node and continue searching further. This type of an approach has advantages to produce smaller *graph* so that the computationally heavy

graph traversal process is reduced. JPS algorithm contains several criterions for the *jump points*, which can be expressed with three separate conditions (see definition 1)

Definition 1.

Definition of *jump point*

y: *jump point*

x: current node

k: step

$\vec{d}$: direction

Node *y* is the *jump point* from node *x,* heading in direction $\vec{d}$, with the following definitions

if *y* minimizes the value *k* such that $y = x + k * \vec{d}$

and one of the following conditions holds:

1. Node *y* is the goal node.

2. Node *y* has at least one neighbour whose evaluation is *forced*

3. $\vec{d}$ is a diagonal move and there exists a node z = $y + k_i * \vec{d}_i$

3.7. PRUNING

The evaluation of a node is based on three properties: direction from the parent (moving direction), adjacent neighbors, and possible obstacles around the node. Even though every node has a maximum of eight nearest neighbors some of these neighbors are irrelevant to the optimal path. It is obvious that going backward cannot be an optimal path, so any previously evaluated node between the parent and current node can be rejected as a potential node to travel. Generally, there is also no reason to conduct any turnings if a location can be reached with a straight trajectory. Therefore, for improving the performance, the algorithm should reject nodes which are certainly not giving an optimal path. The process of rejecting non-relevant nodes is called pruning. Pruning can be seen as a process where it is not known exactly how the optimal path goes but it is known how

it certainly cannot go and those options can therefore be pruned off. There are different rules for pruning depending if the node has obstacles on its nearest neighbors or not. Also depending on the moving direction there are different rules how nodes will be pruned. The *natural neighbors* and *forced neighbor* rules define a minimum set of nodes which are required for searching the optimal path.

3.7.1. NATURAL NEIGHBORS

Applying the pruning rules on the adjacent non-blocked neighbors gives a specific set of neighbor nodes, which are called *natural neighbors*. *Natural neighbors* are a set of traversable nearest neighbors of node *x*, selected by comparing two different options for routing a path. The other path travels from *p(x)* to the nearest neighbor *n* excluding node *x* and the other path travels from *p(x)* to the nearest neighbor *n* including node *x*. Let us assume that the moving direction is horizontal from *p(x)* to *x* and the node *n* is a neighbor of node *x* (Figure 2). Node *n* is natural neighbor because the path from *p(x)* via *x* to *n* is the shortest. A formal definition of *natural neighbors* on a direct (horizontal or vertical) pruning is declared by Equation (1)

$$len(\langle p(x), \dots n\rangle \backslash x \leq len(\langle p(x), x, n\rangle) \quad (1)$$

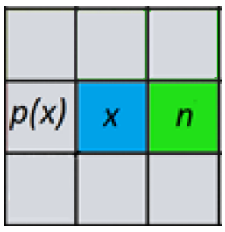

Figure 2. An example how the direct pruning proceeds from *p(x)* to node *x* and rejecting all except the one *natural neighbour* (green coloured node).

In the case of a diagonal move (Figure 3), pruning rules are different. In that case the diagonal move path to node *x* from *p(n)* without visiting node *x* must be strictly dominant as the Equation (2) states.

$$len(\langle p(x), \dots n\rangle \backslash x < len(\langle p(x), x, n\rangle) \quad (2)$$

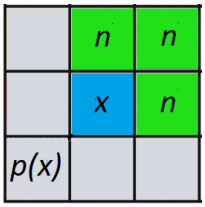

Figure 3. An example how the diagonal pruning is proceeding from *p(x)* to node *x* and rejecting all except the three *natural neighbours* (green coloured nodes).

3.7.2. FORCED NEIGHBORS

When some of the nearest neighbours of node *n* are blocked, additional nodes around *n* are required to be considered as member of a potential path (see definition 2). These nodes are called *forced neighbors*. It can be seen that the algorithm is forced to consider these nodes and directions. A f*orced neighbor* is an essential concept in the JPS algorithm because it ensures the optimality when obstacles are along the search path. It also minimizes the required set of nodes to evaluate, when only potential nodes for the optimal path are considered. Let us assume that the moving direction is horizontal from *p(x)* to *x* and the node *n* is a neighbor of node *x* (Figure 4). Additionally, there is also a blocked neighbor node n_b. Location of the node n_b, is such that *forced neighbor* node n_f is required to evaluate. (Figure 5) illustrates case of diagonal pruning with *forced neighbor*.

Definition 2.

A *forced neighbor n* fulfills two requirements 1. Node *n* is not a natural neighbor of node *x*. 2. len($\langle p(x), x, n \rangle$) < len($\langle p(x), \ldots n \rangle$\x)

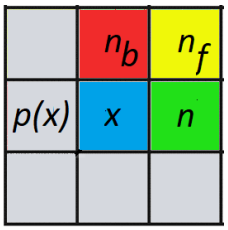

Figure 4. Example how the direct pruning with *forced neighbour* proceeds from *p(x)* to node *x* and rejects all except the *natural neighbour n* and the *forced neighbour* n_f.

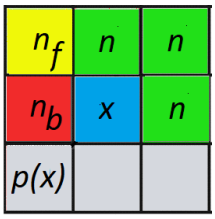

Figure 5. Example how the diagonal pruning with *forced neighbour* is proceeding from *p(x)* to node *x* and rejecting all except the three *natural neighbours n* and the *forced neighbour* n_f.

3.8. IDENTIFY SUCCESSOR

JPS algorithm can be thought as an extension of A* algorithm. A* algorithm is a pure *graph traversal* algorithm which takes a weighted graph as an input data. It conducts the actual pathfinding by comparing weight values (distances) between the vertices. JPS algorithm runs parallel with the *graph traversal* algorithm as it evaluates and selects nodes based on the set of rules so that the actual graph would be smaller than what pure

A* algorithm can produce. For that purpose, JPS algorithm introduces a function which is called *identify successor*. It can be thought as the main algorithm/function of JPS (Figure 6)

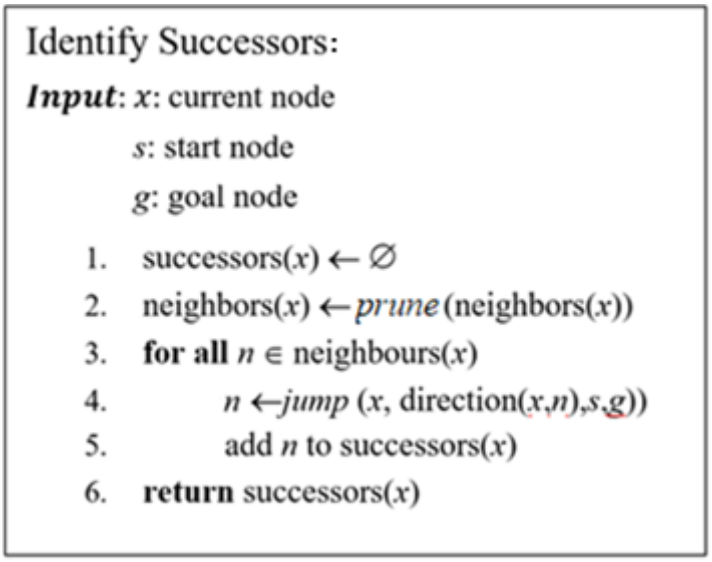

Figure 6. Pseudocode of *Identify successor*

Identify successor constructs a list of nodes for the *graph traversal* algorithm. It takes three input parameters: current node *x*, start node *s* and goal node *g*. The function returns a list of nodes which fulfill the requirements of the *jump point*.

It is noticeable that the code of *identify successor* is quite short and simple as most of the functionalities are hidden in the functions like the *prune* and *jump.* The algorithm also utilizes direction information (function *direction)* which is the main difference compared with A* algorithm. The detailed operation of *identify successor* function will be explained later when the 3D version of *identify successor* will be descriped as the function will be practically the same on the both versions.

3.9. JUMP FUNCTION

Most of the heavy computation is conducted by the *jump* function (see Figure 7). It is noticeable that the *jump* function is recursive, which eventually returns a node or null. Operation of the *Jump* function can be expressed formally by the equation $y = x + k * \vec{d}$. On the equation, variable x represents the current node. The factor $k \in \{1, 2, 3...n\}$

represent a value how many steps the function has taken (recursive calls). The variable $\vec{d}$ represents the current moving direction. The result of the equation is the variable *y* which can be a *null* or *jump point*. To determine if the node *x* is a *jump point* it must fulfill certain requirements of the *jump point* and at the same time value of *k* must be minimized. In practice, it means that the *jump* function must return the first node that satisfied the requirement of the *jump point*.

There are exactly three cases when *jump* function can return a node. It is noticeable that all nodes that *graph traversal* algorithm receives are *jump point* nodes. See Figure 7 for pseudocode of the *jump* function. The *step* function (line 1.) returns a node *n* which is neighbor of *x* on direction $\vec{d}$.

Jump function:

Input: *x*: initial node

$\vec{d}$: direction

s: start node

g: goal node

1: $n \leftarrow step(x, \vec{d})$
2: **if** *n* is an obstacle or out of grid **then**
3: **return** *null*
4: **if** *n* = *g then*
5: **return** *n*
6: **if** ∃ *n'* ∈ neighbors(*n*) s.t. *n'* is forced **then**
7: **return** *n*
8: **if** $\vec{d}$ is diagonal **then**
9: **for all** *i* ∈ {1,2} **do**
10: **if** *jump*$(n, \vec{d}_i, s, g)$ is not *null* **then**
11: **return** *n*
12: **return** *jump*$(n, \vec{d}, s, g)$

Figure 7. Pseudocode of *jump* function

The first case which satisfies the definition of *jump point* (see Figure 7) is evaluated on line 4. If the node is equal with the *goal* node, it will return the *goal* node as the path has

been found. This can occur only once during the search when the search has succeeded, and no further search is required.

The second case is evaluated on line 6 if the current node has any neighbor node which fulfilled the requirement of the *forced neighbor*. This case can occur relatively frequently when the search process is bypassing the obstacles on a space.

The third case is evaluated on lines 8…11. The first condition is that the search direction must be diagonal, if so, then *jump* function is called sequentially with horizontal and vertical direction parameters. If any of those recursive method calls will return non-null, then the node will be the *jump point*. It is noticeable that *jump* function with diagonal direction parameters always returns to two straight-direction *base cases*. The mentioned base cases are evaluated on line 12. Any other cases like searching off the limits or entering non-walkable area, the function will return *null.*

4. JPS AS A 3D-VERSION

To identify the requirements of 3D JPS adaptation it is necessary to understood how the extension to the third dimension generally affects the pathfinding problem and how it affects the principles of the JPS algorithm. Pathfinding problems are then analyzed from different theoretical perspectives to identify algorithmic challenges and to discern what is the essence of the pathfinding problem.

The original JPS algorithm is based on theorems, lemmas and definitions that define the rules how the algorithm operates on a 2D space. The principles of the JPS algorithm have been proven to be functional in theory and practice to find optimal path on 2D grid space. However, one cannot assume that all rules for 2D space are valid or even applicable on a three-dimensional cubic grid space. It is still conceivable that some of these rules can be defined generally enough to make it possible to extend the JPS to three-dimensional space.

Extending 2D space to 3D space has a significant impact on the complexity of the pathfinding process. The effect is more fundamental than the additional coordinate value and the expanded search space. To clarify the difference, one can think of an example: how to bypass an obstacle in a two-dimensional space versus a three-dimensional space. There is a crucial difference: a two-dimensional plane allows only two options for the path; to pass it by from either side of the obstacle. Regardless of the size or shape of obstacle, the number of options remains the same. In a 3D space, the obstacle can be bypassed from any side with many distinct locations. From the perspective of performance, the effect of granularity has also a significant impact on the 3D space. Finer granularity adds more options on how a path can bypass obstacles. Even though the set of options is finite, computationally the 3D pathfinding is significantly more complex and demanding than the two-dimensional pathfinding.

From the perspective of a path, there are some similarities between the 2D and 3D spaces. The path is traveling and changing its direction whenever it is required for preserving the optimality. It is intuitively obvious that any (optimal) path except a straight line on a

discretized space requires one or more turning point(s) between the start and goal point. This requirement remains whether the path travels in two or three-dimensional space. The turning points represent allowed non-blocked locations for the path on space or plane. The third dimension adds a degree of freedom to the turning points. If there is an algorithm which can solve those turning points, then the turning points must be equal to the *jump points*. This holds regardless of the dimensionality.

From the perspective of the *graph traversal* algorithm, the essence of a *jump point* or any other point is just a plain *node* on a graph, without location information. The definition of a *node* is independent from the dimensions. Therefore, the concept of a *jump point* is also applicable in both two and 3D spaces.

It is also possible to identify other basic concepts on JPS algorithm like the definition of the *nearest neighbour*. Basically, it represents the connectivity of a node. In theory, connectivity can be selected freely and in practice, it must establish options for potential paths. Therefore, a practical limitation is that in a 3D space minimum connectivity is required to cover directions off all coordinate axes. To establish a smooth and optimal path, connectivity should also include the diagonal directions. Generally, on a grid-based system, connectivity is usually chosen to cover all adjacent neighbors which has at least one common connection point. In a 3D cubic grid-space the number of adjacent neighbors is at maximum of 26.

Pruning is an essential concept on JPS algorithm. The core idea of pruning is that during the search of an optimal path, one can identify choices which can possibly lead to the optimal path and reject the choices which cannot do it. Pruning reduces the set of nodes which are required for finding the optimal path. In two-dimensional space, pruning can drastically reduce the amount of feasible options. It can also be assumed that in 3D space, optimal path can only have limited amount of choices. Therefore, the idea of pruning is applicable to a 3D space.

4.1. PATHFINDING FROM THE PERSPECTIVE OF SETS

For modelling the pathfinding problem, a graph is widely used as it can represent enough information for creating a simplified abstract model of environment. That model without coordinates is enough for the *graph traversal* algorithm to solve the path. However, a pathfinding problem is generally a selection problem where the algorithm is selecting a special set of nodes (path) from a sets of nodes (space). The selection problem is challenging as node sets are usually large. By using sets, one can create a model for emphasizing the role of selection on a pathfinding problem. This high-level abstraction model is also useful in demonstrating the favorable qualities on the pathfinding algorithm.

It is obvious that any optimal path is the outcome of a specific selection process. It can be demonstrated with this example: Let us assume that one has a finite, discretized space D^n. As a result of the discretization, the space is divided into a set of locations. The locations are called nodes, so one can denote that set **A** includes all the nodes on space D^n. During the pathfinding process, some finite number of nodes are evaluated. One can denote that set **B** includes all the nodes that are required to be evaluated for finding the shortest path. One can also assume that the pathfinding process was successful, so there is also set **C** which includes all the nodes on the found path.

A theoretical general relation between those sets can be expressed: $\mathbf{B} \subseteq \mathbf{A}$. It is obvious that set **A** becomes large when the space or granularity or both increases. However, one can assume that almost in any typical pathfinding problem, it is not necessary to evaluate all the nodes in the space for finding a path. Therefore, one can denote: $\mathbf{B} \subset \mathbf{A}$ thus $|\mathbf{B}| \leq |\mathbf{A}|$. It can be also assumed that for any non-trivial pathfinding problem, a *graph traversal* algorithm must evaluate a remarkably larger number of nodes compared to the number of nodes which belong to the shortest path. The pathfinding process is an iterative process. Therefore, set **B** is inevitably evolving and increasing during the pathfinding process. Limiting the growth of set **B** can be done by using heuristic methods. By estimating the distance to the goal, the search is directed towards the goal. Despite the general usefulness of heuristic methods, in practice, the accuracy of those methods is limited.

It is worth noting that with the optimal path, the set of **C** is selected by minimizing total sum of node weight values. therefore, the minimum size of set **C** can vary. However, it is typically smaller than set **B**. Thus, when the size of set **B** gets closer to the size of set **C**, it means that more nodes expanded by the pathfinding algorithm will eventually belong to the shortest path. In general, it can be denoted $\mathbf{C} \subset \mathbf{B}$ and $|\mathbf{C}| << |\mathbf{B}|$. This means that the algorithm is more efficient for selecting the valid nodes, which is naturally favorable.

In summary, when thinking of the pathfinding problem from the perspective of sets: **A**, **B**, **C** and the relations between these sets. It comes clear that the essence of graph-based pathfinding problem is how to efficiently select the smallest possible set **B** from set **A**, so that ideally $|\mathbf{B}| = |\mathbf{C}|$. In practice, such situation is very unlikely. JPS algorithm addresses this problem with *jump point search*, it uses additional functions to reduce size of set **B**. It is also worth noting that even though one could minimize the size of set **B** using some additional algorithm, those algorithms are also processes that require some computational recourses. The overall efficiency of an algorithm is the total sum of time complexities in selecting set **B** and **C**. Therefore, minimizing set **B**, is not necessarily improving the performance of pathfinding process.

4.2. PATHFINDING ON THE PERSPECTIVE OF GRAPHS

A graph is a comprehensive representation of the pathfinding problem. The information is enough for the *graph traversal* algorithm to solve the path. In the pathfinding process, the graph presents the environment on a higher abstraction level than maps of the physical world. Information is presented without layout information and obstacles. For the *graph traversal* algorithm, the location or layout itself is a non-relevant property. Pathfinding process is conducted purely by comparing distances between nodes and, therefore, the only required information is the travel costs to a node. That information can be stored on a single value in a node and, thus, a weighted graph is enough to express the distances between nodes. From the perspective of a graph, the pathfinding problem can be considered as a combinatorial problem. The algorithm tries to find such a variation from the set of interconnected nodes so that the total sum of the weight values is minimized.

When refer to a space or plane where the pathfinding is conducted, one can assume that all the traversable areas are interconnected. Therefore, the whole space can be recognized as one graph *G*. it includes all the nodes and each of them has a connection to its neighbor nodes. There is no fundamental restriction in having a permanently impassable route from one space to another space. However, algorithmically thinking, those would be separated and individual graphs. Additionally, one cannot have a meaningful (solvable) pathfinding problem between separate spaces (graphs).

The pathfinding process is iterative as the set of candidates for an optimal path is evolving during the process. From the perspective of graph, it means that usually a large number of nodes and weights must be evaluated to find out the smallest combination of those. That process can be seen as creating a graph G_1, which is a sub-graph of graph *G*. In the *graph traversal* algorithm, this sub-graph consists of the nodes that belong to *openlist* and *closedlist*. Once the *graph traversal* has found the shortest path, the path is also graph denoted G_2 which is also sub-graph of *G*.

From the perspective of *graph traversal*, a larger physical size of the traversable environment requires commonly more nodes to represent it and, therefore, pathfinding becomes more complex. In general, the more there are nodes and connections between them, the more options there are and, therefore, pathfinding becomes computationally heavier. The performance of a *graph traversal* algorithm is significantly affected by the size of the graph, as each node is required to be evaluated by the same method.

JPS algorithm has a relative effective technique to mitigate the above problem. However, the geometry of the environment still influences on the performance of JPS. JPS algorithm benefits on the existence of open spaces where JPS can select a single optimal path efficiently without considering other optimal paths (symmetry breaking). However, the performance might decrease in extremely large open environments. On those circumstances, the search might take long time as *jump* function recursion continues until it finds a *jump point* or returns null. If the environment has a lot of obstacles, generally more *jump points* are generated, and each *jump point* is also source for a search for the next *jump point*. Therefore, the size of *openlist* increases which has generally negative effect on performance of *graph traversal* algorithm.

However, *jump point* is a general definition; it refers of nodes where the optimal path might travel. It is noticeable that each *jump point* is always defined from the perspective of its parent node. The obvious challenge for the 3D-modification is how to apply the principles of the original JPS so that the optimality of the path will remains and at the same time the performance will be at a satisfying level. Those qualities define how applicable is the pathfinding algorithm; To be a meaningful algorithm, it should be able find path in time which is competitive to other algorithm with same qualities.

4.3. ANALYZING THE PRINCIPLES OF JPS

The structure of JPS is more complex compared with A* its adaptation to 3D requires more than just expanding the coordinate system to 3D. The problem here is that the original version of JPS is tightly coupled on a 2D space. Many rules of JPS algorithm would become ambiguous in 3D space. Especially because the third dimension extends the number of possible separate directions from 8 to 26. On JPS, the direction information is an essential property for evaluating the optimal path. It is intuitively obvious that also in a 3D space, optimality requires that all possible combinations of the 26 directions must be considered.

JPS uses the term: the *diagonal direction*, for defining the pruning rules. The concept of diagonal direction is clear definition on a 2D space, but it becomes ambiguous on higher dimensions. By intuition, one might assume that the diagonal direction in a 3D space (x,y,z) is line from the origin to any location with three non-zero end coordinates. However, it is also true that line with even two separate coordinates on a 3D space already defines a diagonal direction. Obviously, there are numerous possible choices, and therefore a systematic method is required to generalize the concept of those rules.

4.4. JUMP POINT AND PRUNING OPERATIONS

The pruning rules have a significant impact on the optimality and performance of the JPS algorithm. JPS algorithm defines separate pruning rules for straight and diagonal cases.

Equations for defining the direct (Equation (1)) and the vertical pruning (Equation (2)) are nearly identical (see chapter 3.7.1). However, there is one significant difference which is related to the dominance constraint of the paths. To understand the underlaying reasons for these differences, it is necessary to analyze the JPS algorithm and its pruning concepts without coordinate system related properties.

For a uniform square grid, the pattern of neighbor nodes is always similar and there is the same number of the nearest neighbors. An obvious difference is that in a diagonal direction the minimum moving distance is $\sqrt{2}\ g$, where g is the straight distance of the grid. That is a significant difference, as the pruning rules are based on path length comparison.

To clarify the meaning of distances, it is necessary to revisit the definition of the *jump point*: $y = x + k * \vec{d}$. Equation of the *jump point* defines that the node y is a *jump point* only if the factor k is minimized and at the same time node y satisfies the requirements of *jump point*. On that equation, x represents the current node and the moving/stepping direction is denoted with $\vec{d}$. Those *jump points* represent the turning points of the path, so it is necessary to find the next *jump point* as close as possible to the current node x. That is the necessary requirement as overestimating distance to the next turning point could destroy the optimality [10].

The above equation of the *jump point* forces that *jump points* are correctly found as close as possible. That requirement must hold regardless of the direction where the algorithm is stepping. Node x is a fixed parameter and cannot be changed as it represents the current location from where the counting starts. The only parameter which can be chosen is moving direction $\vec{d}$ and it also affect the distance between adjacent nodes. The distance is 1, if $\vec{d}$ is straight and value $\sqrt{2}$, if $\vec{d}$ is diagonal. From the perspective of minimizing, it implicates that straight directions $\vec{d}$ would give smaller value for the whole equation. Therefore, if possible, straight moving directions are always chosen before diagonal moving directions. Naturally, selection requires that nodes in the directions $\vec{d}_1$ and $\vec{d}_2$ are traversable and not out of boundaries. If the *jump* function is called with diagonal

direction parameter $\vec{d}$, instead of using that direction, the *jump* function is first called with horizontal and vertical direction $\vec{d}_1$ and $\vec{d}_2$.

Equation (2) (see chapter 3.7.1) defines the strict dominance constrain on the JPS algorithm JPS algorithm defines *jump* function (Figure 7) where the lines 8...10 causes that diagonal direction movement is first sweeping horizontal and vertical directions and then stepping one unit on diagonal direction $\vec{d}$. The whole cycle of the diagonal *jump* function call in a two-dimensional space without obstacles is shown on (Figure 8) where the gray node is the parent and the blue nodes are already visited nodes. The first (Figure a) shows how the diagonal direction $\vec{d}$ is divided to horizontal jump function calls with direction $\vec{d}_1$ (Figure b) and to vertical function calls with direction $\vec{d}_2$ (Figure c).

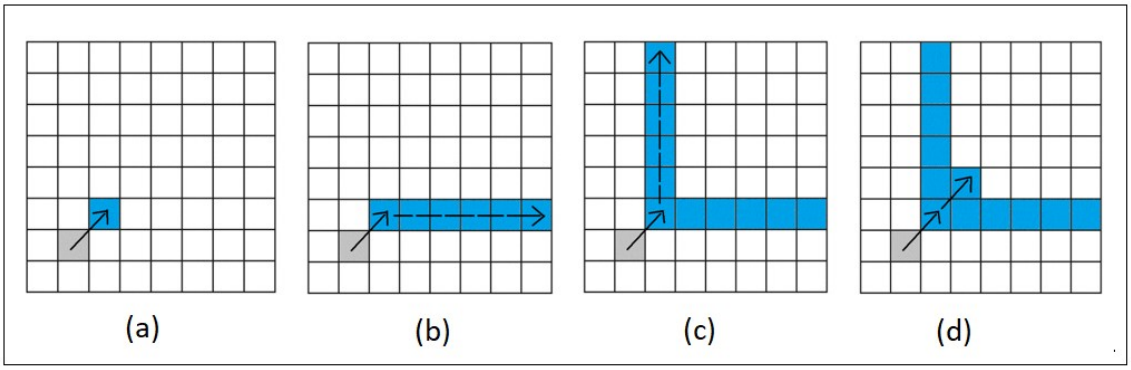

Figure 8. A diagonal *jump* function call is sweeping both horizontal and vertical lines before taking a diagonal step ahead.

To sum up, it is obvious that the *natural neighbor* pruning rules on JPS algorithm are not directly related to the angle of the moving direction. Separate rules are required when the different directions cause different distances between the nodes. In addition, distance is a pure scalar value and not related to the coordinate-system, therefore this idea can be generalized to a 3D space. Even though 3D space adds 18 new angles on the space, compared with 2D spaces, it is obvious that in a 3D space, there is no need to consider those as separate directions but rather as a distance. In a 3D space with cubic grids, the minimum straight distance between two adjacent nodes can be denoted with a. Therefore,

the maximum distance is 3D diagonal, calculated by equation $\sqrt{a^2 + a^2 + a^2}$. Thus, if the minimum distance is 1, adding the third dimension adds one new distance $\sqrt{3}$ between neighbor nodes.

5. NOTATIONS AND DESIGN OF THE JPS ALGORITHM

A guideline for the 3D modification of the JPS algorithm was to preserve the core structure equivalent to the original 2D version. By that approach, the functionality of the new algorithm I called JPS_3D can be presented using similar notations and style as before and the modified structures are clearly identified. Functions which have been adapted to the 3D space, are denoted by *"_3D*" endings, for separating those from the original JPS algorithm. Data structures like lists which are not modified, use original JPS algorithm notation.

The modification is fundamentally equivalent with the original JPS algorithm extending the same functionality to a 3D space. It is inevitable that the structure and implementation of 3D version become more complex than the original JPS algorithm. Actual implementation JPS_3D with *C#* language is relatively large containing hundreds of lines, and it is not included on this thesis. The reason for this is that the source code is less informative without knowledge of the syntax/semantics. Therefore, the essential and complicated concepts, like the *jump* function will presented by pseudo codes. The pruning rules and other key concepts are presented by the mathematical definitions, additionally illustrative figures are used for visualizing those.

5.1. IDENTIFYING THE SUCCESSOR NODES IN 3D

Identify_successor_3D of JPS_3D has similar purpose as the *identify successor on* JPS algorithm. It is a straightforward modification from the JPS algorithm. *Identify_successor_3D* is the main algorithm and its function is to search and return jump points (nodes) for the *graph traversal* algorithm. It should be noted that this JPS_3D algorithm does not present the complete pathfinding solution. Complete pathfinding process requires still the use of *graph traversal* algorithm. Advantage for using graphs on pathfinding algorithm is that they are non-depended on the actual layout or coordinate

system. In this case it means, that the core algorithm *Identify_successor_3D* can be similar on both JPS and JPS_3D algorithms. There are no significant structural differences, only the names of the methods are updated for the 3D version, see Figure 9 for pseudocode.

Identify_succesors_3D:

Input: x : current node

s : *start* node

g : *goal* node

1: *successors(x)* ← ∅

2: *neighbors(x)* ← *prune_3D(neighbors(x))*

3: **for all** $n \in neighbors(x)$

4: $n \leftarrow$*jump_3D(x,direction(x,n),s,g))*

5: add *n to successors(x)*

6: **return** successors(x)

Figure 9. Pseudocode of the *Identify_successor_3D*.

Identify_successor_3D explained line by line:

Line 1. JPS_3D algorithm initializes one-dimensional successors*(x)* array. The Purpose of *successors* is to supply nodes for the *graph traversal* algorithm. The 3D modification does not require altering this data structure and it remains same with the original version.

Line 2. *neighbors'* array: This array stores the set of pruned, neighbor nodes. Function *Prune_3D*, takes two input parameters: node x and the list of neighbors of x. Three-dimensional modification does not affect the structure of *neighbors.* Also, the *prune_3D* function is similar with the *prune* on JPS algorithm. However, the *prune_3D* is modified

so that it can include all 26 nearest neighbors on a 3D space. The actual pruning rules are presented on the chapter 5.2.

Lines 3,4,5: The lines perform similar task as the original JPS algorithm. The loop structure: *for all,* iterates the list of the neighbors until the list is empty. During each iteration round, *jump_3D* is called with four input parameters: current node *x*, *direction*, the *start* node and the *goal* node. The node *x* represents a current location on space. Function *direction* takes two input parameters: the current node *x* and node *n.* The function returns direction as a delta value, represented by the three parameters: Δx, Δy, Δz. Function *jump_3D* returns *n* or *null.* Returns value of the *jump_3D* function is stored on the *successors list.*

Line 6: After the loop has finished *Identify_successor_3D* algorithm returns the *successors list.*

In conclusion, the *identify succesor_3D* is functionally like the 2D *identify successor* function on the JPS algorithm. The major differences between the algorithms are on the *prune_3D* and *jump_3D* functions. *Jump_3D* is presented with pseudocode and the rules for the pruning are stated by formal definition along with examples.

5.2. PRUNING RULES IN 3D

The pruning rules are based on the evaluating the length of a potential paths. The purpose is to find those directions which can possibly offer an optimal path. Therefore, the pruning is always conducted on a certain direction. A direction on a 3D space is a combination of three separate direction vectors *x*, y and *z.* The case where only one direction vector is non-zero is called *direct pruning*. The case where there are exactly two non-zero direction vectors, is called *two-axis diagonal pruning*. And finally, when are all the three non-zero direction vectors define the direction, there is *three-axis diagonal pruning*.

Every node in a 3D space has 26 adjacent and nearest neighbors (Figure 10). From the perspective of performance, it is necessary to select only a minimum set of nodes which

are potentially on the optimal path. Function: *prune_3D* returns a list of *pruned neighbors* based on the JPS algorithm pruning principles which are now extended to a 3D space. The function takes a node as a parameter and each evaluated node except the *start* node holds also an information about its own parent node. The parent information is required for indicating direction of the movement and keeping track how the path to the current node is travelled. If the node does not have a parent value, all the nearest neighbor nodes are returned. This occurs only once at the beginning of the search process, at the *start* node. JPS_3D algorithm evaluates nodes based on the entering direction to the specific node. In a 3D space possible direction are defined by delta value of the three coordinate axes (*x*, *y*, *z*). Those delta values can have only three discrete values, -1, 0 and 1. Depending on combination of the delta values, there are 26 directions.

Three-dimensional pruning rules require fundamental modification compared to the two-dimensional JPS algorithm. On the two-dimensional space, the path and the obstacles are all on the same one plane and cannot pass each other using third dimension. From the perspective of pruning, the limited degree of freedom limits also possible paths from the node *n* to its nearest neighbors. It is obvious that every possible path is significant as the minimum length of the path is the criteria for an optimal path.

On a 3D space, there are more possible paths from the node *n* to its nearest neighbor nodes. So even the pruning function is processing only a one direction at the time, it cannot be done without considering all the three dimensions. For example, let us assume that the pruning direction goes along the *x*-axis. The space around that axis is three-dimensional and possible occupied with obstacles so there are several possible paths to evaluate. The algorithm must consider paths not only on the *X*-axis direction but also on the *xyz*-space. It means significantly more directions and combinations to evaluate and inevitable pruning rules become more complicated.

The (*x, y, z*) space is discretized in cubic grids by the grid points: $\{x_0, x_1, x_2, x_n \ldots x_{max}\}$, $\{y_0, y_1, y_2, y_n \ldots . y_{max}\}$ and $\{z_0, z_1, z_2, z_n \ldots z_{max}\}$, additionally x_{max}, y_{max} and z_{max} refers to a finite natural number. The shortest possible distances between nearest neighbor nodes varies between $\{1, \sqrt{2}, \sqrt{3}\}$. On the 3D space the pruning can be done using the three separate pruning rules. The used pruning rule depend on the moving direction. In

case the moving direction goes along a single coordinate axis, direct pruning rules are applied. If the moving direction is combination of a two separate coordinate axis, two-axis pruning rules are applied. The third case is if the direction is combination of all three-coordinate axis, in that case three-axis pruning rules are applied. To understand how the 3D pruning rules works it is necessary to analyze each of those with examples.

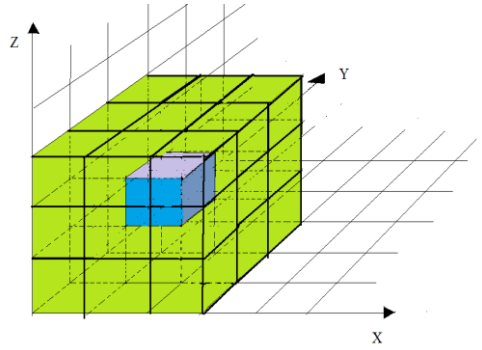

Figure 10. A node (blue) on a center and its 26 adjacent neighbors (green) on a three-dimensional space.

5.2.1. DIRECT PRUNING

Direct pruning rules are applied when the pruning direction goes along *x*, *y* or *z*-axis. The pruning rules for *natural neighbors* are defined by Equation (1) (see chapter 3.7.1) and for *forced neighbors* by the Definition (1). These rules are equivalent with the original JPS algorithm and the only difference is that the distances between the nodes are calculated on a 3D space. Even though pruning is based on arithmetic comparison between the path length, uniform cubic grid makes possible that each node has the similar pattern of neighbor nodes around it with a similar distance. Therefore, computationally heavy arithmetic calculation is not necessary to conduct each time as one can use the previously known path length to each of those neighbor nodes.

Example 1. Direct pruning. Let us assume that node *n* is on the location (x_i, y_j, z_k), and location of parent node *p*(*n*) is (x_{i-1}, y_j, z_k). The location difference between the node *n* and its parent node ($\Delta x = +1$, $\Delta y = 0$, and $\Delta z = 0$), so the pruning direction is direct along

the x-axis, see (Figure 11). Additionally, there are no blocked nodes which would fulfill requirements of *forced neighbors*. Equation 1. (see chapter 3.7.1) defines the *natural neighbors*. Outcome of applying pruning rules is that 25 out of the 26 neighbor nodes are pruned out and only a one node fulfills the requirements of *natural neighbor*.

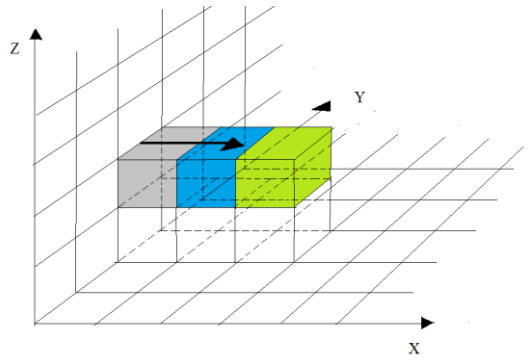

Figure 11. Pruning along the x-axis in a empty 3D space: The parent node (gray) has a single natural neighbor (green). The direction is from the parent node to node *n* (blue), and the pruning returns the one *natural neighbor*

Example 2. Direct pruning *natural neighbors* and *forced neighbor*: Let us assume that node *n* is on the location: x_i, y_j, z_k, and the location of the parent node *p(n)* is (x_{i-1}, y_j, z_k). Immediately adjacent to the node *n*, there is an obstacle on location (x_i, y_j, z_{k-1}), (see Figure 12). The location difference between node *n* and its parent node *p(n)* is Δ(+1,0,0), the moving direction is direct along the x-axis. Equation 1, (see chapter 3.7.1) defines the single *natural neighbor*, it is immediately right from the node *n* on location (x_{i+1}, y_j, z_k). Because one of the *non-natural neighbors* is occupied by an obstacle (red node), rules for a *forced neighbor* (Definition 1.) are required to apply. Therefore, the yellow node in location (x_{i+1}, y_j, z_{k-1}) is the *forced neighbor*.

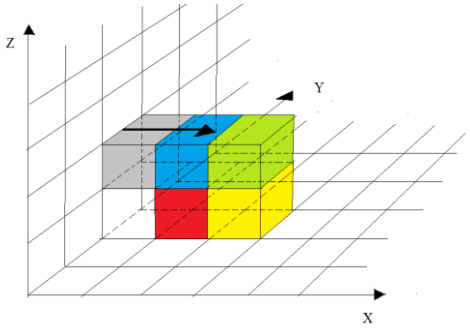

Figure 12. Pruning along the x-axis in non-empty 3D space: The red node is blocked, and the direction is from the parent node (gray) to node *n* (blue). The pruning returns one *natural neighbor* (green) and one *forced neighbor* (yellow)

Example 3. Direct pruning *natural neighbors* and *forced neighbor*: Let us assume that the node *n* is on the location (x_i, y_j, z_k), and the location of the parent node *p(n)* is (x_{i-1}, y_j, z_k). Adjacent to the node *n* there is an obstacle on location (x_i, y_j, z_{k-1}), see Figure 13. The location difference between node *n* and its parent node *p(n)* is Δ(+1,0,0), so the moving direction is direct along the *x*-axis. Equation 1. (see chapter 3.7.1) defines *natural neighbors*. Single *natural neighbor* is immediately right from the node *n* on location (x_{i+1}, y_j, z_k). Because one of the *non-natural neighbors* is occupied by obstacle, rules for a *forced neighbor* must be applied. As a result, the yellow node in location (x_{i+1}, y_{j-1}, z_{k-1}) is determined to be a *forced neighbor*.

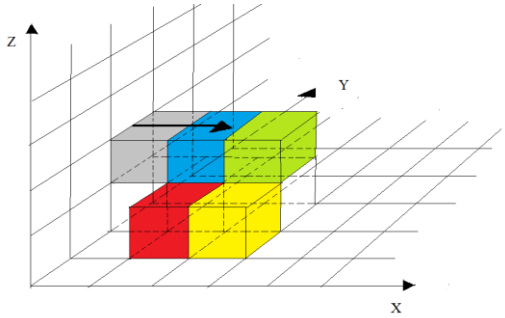

Figure 13. Direct pruning along the x-axis in a non-empty 3D space: The red node is blocked node, and the direction is from the parent node (gray) to node *n* (blue). The pruning returns one *natural neighbour* (green) and one *forced neighbour* (yellow).

Example 4. Direct pruning of *natural neighbor* and multiple *forced neighbors*: Let us assume that node *n* is on the location (x_i, y_j, z_k,) and the location of the parent *p(n)* is (x_{i-1}, y_j, z_k). Adjacent to the node *n* there is two obstacles on locations (x_i, y_{j-1}, $z_{k-1)}$ and (x_i, y_{j-1}, z_k), see Figure 14. The location differences between node *n* and its parent *p(n)*, Δ(+1,0,0), so the move is along the *x*-axis. The *natural neighbors* are defined by the Equation 1. (see chapter 3.7.1) pointing to the location (x_{i+1}, y_j, z_k). Because two of the non-natural neighbor obstacle boxes, rules for a *forced neighbor* must be applied for the both of the obstacles. As a result of pruning yellow nodes on the locations (x_{i+1}, y_{j-1}, z_{k-1}) and (x_{i+1}, y_{j-1}, z_k) are determined to be *forced neighbors*.

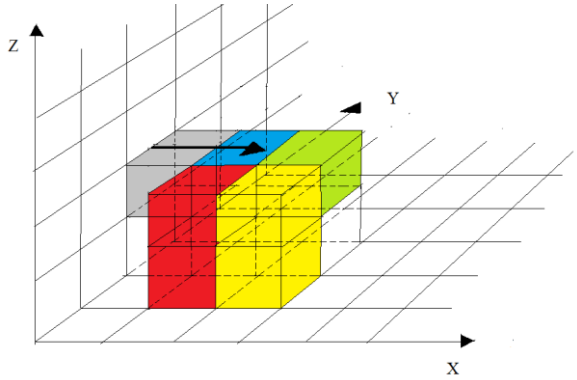

Figure 14. Direct pruning along the x-axis on non-empty 3D space. The red nodes are blocked, and the direction is from the parent node (gray) to node *n* (blue). The pruning returns the one *natural neighbor* (green) and the two *forced neighbors* (yellow)

5.2.2. TWO-AXIS DIAGONAL PRUNING RULES

Two-axis pruning rules are applied when the moving direction is defined with pair of two coordinate axis: *xy*, *yz*, or *xz*. The pruning rules for *natural neighbors* are defined by the Equation 2 (see chapter 3.7.1) and the pruning rules for the *forced neighbors* by the Definition 1. These rules are similar as for the original JPS algorithm; the only difference is that the distances between the nodes are now calculated on a 3D space.

Example 5. Pruning in two-axis diagonal direction for *natural neighbors*. Let us assume that node *n* is on the location(x_i, y_j, z_k) and the location of parent node *p(n)* is (x_{i-1},

y_{j-1}, z_k). The location difference between the node *n* and its parent node *p(n)* is Δ(+1,+1,0) so the moving direction is diagonal on the *xy*-plane, see Figure 15. The Equation 2 (see chapter 3.7.1) defines the *natural neighbor*. Applying the pruning rules return the three nodes which fulfills requirement of *natural neighbor.*

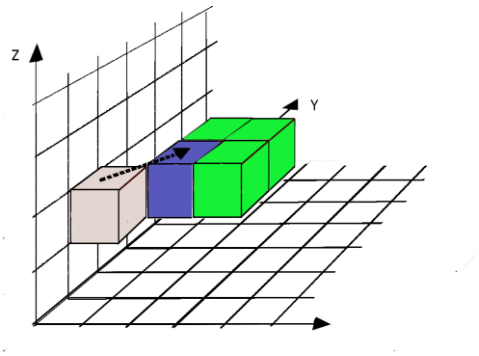

Figure 15. Pruning in the two-axis diagonal direction on an 3D space. The direction is from the parent node (gray) to the node *n* (blue). The pruning operation returns three *natural neighbors* (green).

Example 6. Let us assume that node *n* is on location (x_i, y_j, z_k) and the location of parent node *p(n)* is (x_{i-1}, y_{j-1}, z_k). The location difference between node *n* and its parent *p(n)* is Δ(+1, +1, 0) so the moving direction is diagonal on *xy*-plane, see Figure 16. *Natural neighbors* are defined by Equation 2 (see chapter 3.7.1) Additionally, one of the *non-natural neighbors* is occupied by obstacle (red node), so rules for the *forced neighbor* (Definition 1) are required to apply.

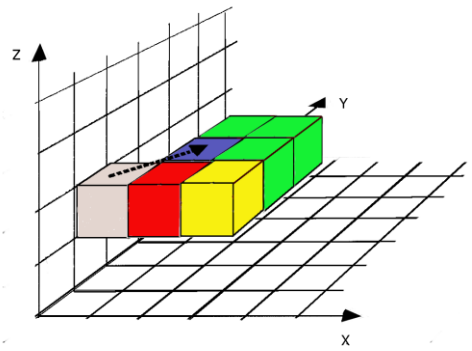

Figure 16. Pruning rule along the *xy*-plane on a non-empty 3D space. The red node is blocked, and the direction is from the parent node (gray) to node *n* (blue), The pruning returns three *natural neighbors* (green) and one *forced neighbor* (yellow).

Example 7. Pruning two-axis diagonally *natural neighbors* with multiple *forced neighbors*: Let us assume that node *n* is on location (x_i, y_j, z_k) and the location of the parent node *p(n)* is (x_{i-1}, y_{j-1}, z_k). The location difference between the node *n* and its parent *p(n)* is Δ (+1, +1, 0). The moving direction is diagonal on *xy*-plane, see Figure 17. *Natural neighbors* are defined by the Equation 2 (see chapter 3.7.1) Additionally, two of the *non-natural neighbors* are occupied by the obstacles (red nodes), so rules for the *forced neighbor* (Definition 1) are required to apply. It is noticeable that even the pruning direction is strictly on the *xy*-plane, the obstacle on the location (x_i, y_{j-1}, z_{k+1}) forces pruning to return *forced neighbor* nodes on the location (x_{i+1}, y_{j-1}, z_{k+1}) This is necessary as the dominant path to those nodes goes always through the node *n*.

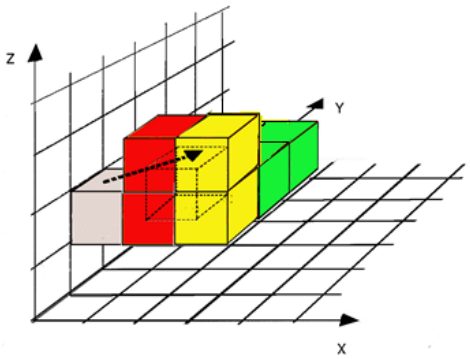

Figure 17. Pruning rule along the *xy*-plane on a non-empty 3D space. The red nodes are blocked, and the direction is from the parent node (gray) to node *n* (the wire frame inside), The pruning returns the three *natural neighbors* (green) and the two *forced neighbors* (yellow).

It is noticeable that each of those *two-axis diagonal pruning* rules are fundamentally similar regardless which of the two-axis diagonal direction is in use. For example, pruning on the *zy*-direction returns *natural neighbors* and *forced neighbor* with the equivalent way (Figure 18). Similarly pruning on the *zx*-direction returns a set of *natural neighbors* and *forced neighbors*. (Figure 19).

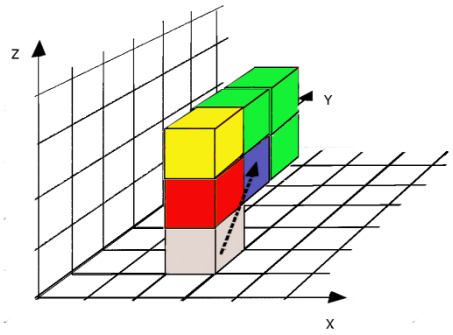

Figure 18. Pruning rule along the *zy*-plane on a non-empty 3D space. The red node is blocked, and the direction is from the parent node (gray) to node *n* (blue), The pruning returns three *natural neighbors* (green) and one *forced neighbors* (yellow).

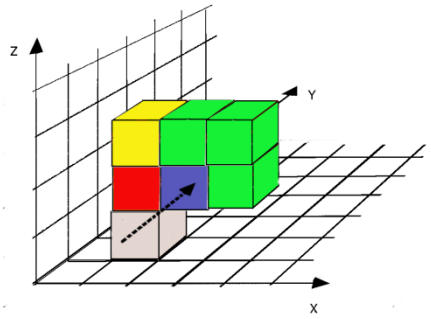

Figure 19. The pruning rule along the *zx*-plane on a non-empty 3D space. The red node is blocked, and the direction is from the parent node (gray) to node *n* (blue), The pruning returns the three *natural neighbors* (green) and the one *forced neighbors* (yellow).

5.2.3. THREE-AXIS DIAGONAL PRUNING RULES

Three-axis pruning rules are applied when the moving direction is defined by a combination of the three coordinate axes. Each *x*, *y* and *z* direction vector must have a non-zero positive or negative value. Possible moving combinations are maximum of eight. The *natural neighbors* are defined by Equation (2) (see chapter 3.7.1) Applying Equation (2) to a neighbor node *n* returns at most seven *natural neighbors*. The neighbors represent all the separate directions how the optimal path on open space can fork from the node *n*. If node *n* has any *non-natural neighbor* nodes which fulfil the requirement of *forced neighbor* (Definition 1) then pruning rule of *forced neighbor* must be applied.

Example 8. Let us assume that node *n* is on location (x_i, y_j, z_k) and the location of the parent node *p(n)* is $(x_{i-1}, y_{j-1}, z_{k-1})$. The location difference between node *n* and its parent node *p(n)* is then Δ(+1,+1, +1) and the moving direction is diagonal on a *xyz*-space (Figure 20). Equation (2) (see chapter 3.7.1) defines *natural neighbors*.

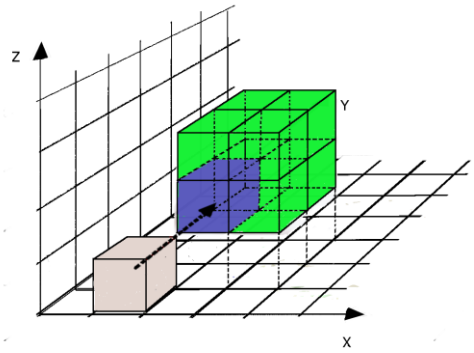

Figure 20. *Natural neighbors* on a 3D space, and the direction is from the parent node (gray) to node *n* (blue). The pruning returns seven *natural neighbor* nodes (green).

Example 9. Let us assume that node *n* is on the location (x_i, y_j, z_k) and the location of parent node *p(n)* is ($x_{i-1}, y_{j-1}, z_{k-1}$). The location difference between node *n* and its parent node *p(n)* is Δ(+1,+1,+1) so the moving direction is diagonal on *xyz*-space (Figure 21). Equation (2) (see chapter 3.7.1) defines *natural neighbor*. Additionally, one of the

non-natural neighbor nodes is occupied by the obstacle (red), so the rules for *forced neighbor* (Definition 1) are required to apply.

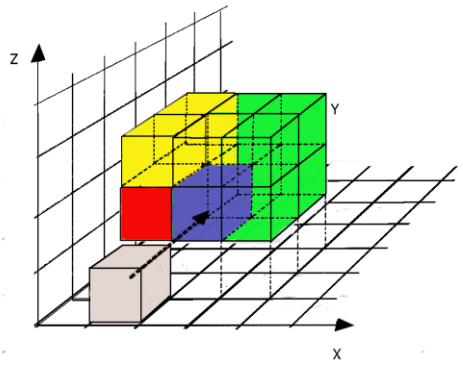

Figure 21. Pruning diagonally along the *xyz*-space in a non-empty 3D space. The red node is blocked, and the direction is from parent node (gray) to node *n* (blue). Pruning returns seven *natural neighbors* (green) and the three *forced neighbors* (yellow).

In a conclusion, the set of the pruning rules defines how to evaluate any node on a 3D space using information about the parent of a node and the space around the evaluated node. Applying the pruning rules to node *n* returns a set of nodes so that path from *p(n)* via *n* to any of those nodes is optimal path.

5.3. JUMP_3D FUNCTION

Function *jump_3D* is a fundamental part of the JPS_3D algorithm as it provides *nodes* to the main algorithm and most of the computational effort is conducted inside the *jump_3D* function. The function calls also the pruning functionalities when required. Three-dimensional environment requires significantly more evaluation and therefore the *jump* function is more complex than the original *jump* function. The *Jump_3D* function is recursive, calling itself with a particular set of parameters.

The main purpose of the function is to process *nodes* in a 3D space and return the node when it fulfills requirement of the *jump point* otherwise the search was unsuccessful, and it must return null. *Jump points* on a 3D space have an equivalent definition with the original JPS algorithm, it represents a possible turning point for the optimal path. It is

noticeable that the *jump_3D* function searches a node candidate for the possible path, but the function is not calculating the actual path. *Jump_3D* function takes several input parameters, including current node, direction parameter, *start* and *goal* nodes. Normally, *jump_3D* function requires the direction parameter based on the parent node. Only the first call of *jump_3D* function is the special case as it does not have information about the parent node, therefore at the beginning of the search process it is mandatory to evaluate all the possible directions from the *start* location as each of those directions is equally possible. The pruning around the *start* node returns all the (if those nodes exist) 26 nearest neighbors. Every function call after that uses direction information evaluated from the parent node.

A parent node is a refence point for the next *jump point* search and each returned node except the *start* node must have it. The path from parent node *p(n)* to *n* represents optimal path. It is noticeable that even *len(p(n), n)* is always optimal, there is no guarantee that *len(p(p(n)), p(n), n)* is all the time optimal. During the search process a new and shorter path to node *n* might be found via another node, therefore the value of *p(n)* will be updated when necessary. There is also no requirement that a parent node must be the adjacent neighbor of the node unlike the A* algorithm requires. A parent node will be used to calculate a direction parameter for the upcoming *jump_3D* calls. Because function *jump_3D* is a recursive function, it must call itself with a new direction parameter set until it founds a *jump node* or null. Those *jump points* will be new reference points for the main algorithm (*Identify_successor_3D*). Typically, a single *Jump_3D* call can expand to numerous separate recursive calls. The expand to multiple directions so that the path from the parent node to the returned *jump point* is always optimal. This autonomous self-guiding is achieved by the inner structure of *jump_3D* function. The structure of the *jump_3D* function enables that it calls itself with the proper direction parameters so that the call order prefers always the currently best path. Whenever it is possible, *jump_3D* function prefers direct moving directions over two-axis diagonal moving directions, and two-axis diagonal moving directions over three-axis diagonal moving directions. When the function founds the *goal*, or a *jump point* it returns that node, otherwise the search goes off the limits or hits an obstacle and must return null (see Figure 22). This moving order is valid also in the cases where the optimal path could be found preferring a diagonal direction first (see Figure 23).

It is noticeable that *jump_3D* function can search and return *jump points* in a non-sequential order from the nearest to the furthermost. Even function *jump_3D* uses a direction parameter to step further, it does not have to find the closest *jump point* first, measured by Euclidian distance. It is possible that the function *jump_3D* can first return a node which distance for instance is 15 from the origin, even if later the closest *jump node* is founded with distance only 2.8. This arrangement will not lead to overestimation and ruining the optimality of path as long the order of search direction follows rules: *direct*, *two-axis diagonal* and *three-axis diagonal*. Using a *parent relation*, the algorithm has built in feature to correct shortest distance value to any *jump point* if the new and shorter path is found via other *jump point*.

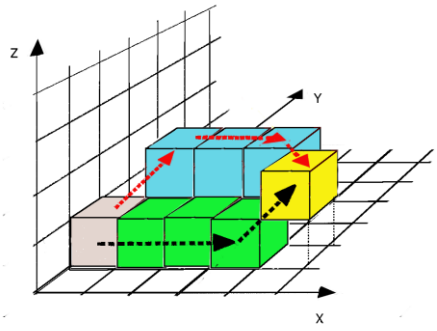

Figure 22. An example how different moving order in a 3D grid space affects the path length. The gray node is the *start* node and yellow node is the *goal* node. The green colored nodes represent the optimal path which prefers direct moves over the diagonal moves. The cyan colored nodes represent the non-optimal path which prefers *xyz*-diagonal and *xy*-diagonal moves over the direct moves. It is noticeable that on the both cases the number of the included nodes are the same (4), even the optimal path can be found only by preferring direct moves over diagonal moves.

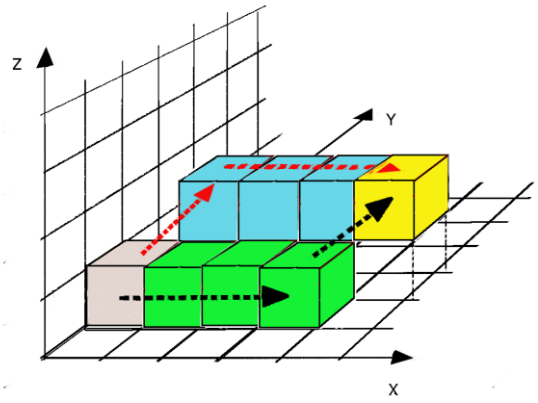

Figure 23. Example how preferring a direct moving direction can preserve the optimality also in the case where a diagonal direction first, could give an equal optimal path.

5.4. NOTATIONS AND THE PSEUDOCODE

Function *jump_3D* uses additional terms to express three-dimensional directions a. Generally, a direction in space is denoted by $\vec{d}$ which can refer to any allowed arbitrary direction on the space. A direction consists of three components; the three coordinate axes are denoted by variables $\vec{d}_x$, $\vec{d}_y$ and $\vec{d}_z$. Each component can have positive or negative factor depending if the direction is to right or left side from the origin. The diagonal direction on a *xy*-plane is denoted by $\vec{d}_{xy}$. The diagonal direction on a *xz*-plane is denoted by $\vec{d}_{xz}$. The diagonal direction on a *yz*-plane is denoted by $\vec{d}_{yz}$. The diagonal direction on a *xyz*-space is denoted by $\vec{d}_{xyz}$.

The *jump_3D* function (Figure 24) is an extended version of the original JPS *jump* function. Each call of *jump_3D* propagates several other recursive calls for reducing the complexity of the original problem. For instance, a single function call with *xyz*-diagonal direction parameter is divided *x*, *y*, *z*, *xy*, *xz* and *yz*-direction calls. Furthermore, those *xy*, *yz* and *xz*-direction calls will be divided correspondingly to separate *x*, *y* and *z*-direction calls.

Figure 25. demonstrates the operation of *jump_3D.* While a single *jump_3D* function call will spread to multiple separate calls. In order the keep the picture, clear this example

shows only a one step on *xyz*-direction. The *current node* is *x* (the blue node) and the *direction* parameter $\vec{d}$ is *xyz*-diagonal. For simplicity let us assume that node *x* is not the *start* node and the space in the current direction is empty without the *goal* node. Then the three-axis diagonal direction parameter satisfies *if statement* condition (line 23). Inside the *if statement* there is a recursive call of *jump_3D* function using only a single direction component *x*, *y* and *z* (lines 24-29). The function calls will instantiate a new copy of the function with proper parameters. Next, the algorithm does new recursive function calls for the three separate directions (*xy*, *xz*, *yz*) using the corresponding components from the *xyz*-direction parameter. Each function call will instantiate a new copy of the function with a proper parameter. Lines 8, 13 and 18 will correspondingly have two-axis diagonal direction *if statement* for its condition will be fulfilled. A new recursive call using a separately both of the diagonal direction components will be then done. Line 36 is the final statement which does the direct single-axis *Jump_3D* call (the base case).

jump_3D function:
Input: *x*: initial node, $\vec{d}$: direction, *s*: start, *g*: goal

1: $n \longleftarrow step(x, \vec{d})$
2: **if** *n* is an obstacle or out of grid **then**
3: **return** null
4: **if** *n* = *g then*
5: **return** *n*
6: **if** *n'* ∈ neighbors(*n*) s.t. *n'* is forced **then**
7: **return** *n*
8: **if** $\vec{d}$ is diagonal on *xy*-plane **then**
9: **if** *jump_3D (n,* $\vec{d}_x$*, s, g)* is not *null* **then**
10: **return** *n*
11: **if** *jump_3D* (*n,* $\vec{d}_y$*, s, g)* is not *null* **then**
12: **return** *n*
13: **if** $\vec{d}$ is diagonal on *xz*-plane **then**
14: **if** *jump_3D* (*n,* $\vec{d}_x$*, s, g)* is not *null* **then**
15: **return** *n*
16: **if** *jump_3D* (*n,* $\vec{d}_z$*, s, g)* is not *null* **then**
17: **return** *n*
18: **if** $\vec{d}$ is diagonal on *yz*-plane **then**
19: **if** *jump_3D* (*n,* $\vec{d}_y$*, s, g)* is not *null* **then**
20: **return** *n*
21: **if** *jump_3D* (*n,* $\vec{d}_z$*, s, g)* is not *null* **then**
22: **return** *n*
23: **if** $\vec{d}$ is diagonal on *xyz*-space **then**
24: **if** *jump_3D* (*n,* $\vec{d}_x$*, s, g)* is not *null* **then**
25: **return** *n*
26: **if** *jump_3D* (*n,* $\vec{d}_y$*, s, g)* is not *null* **then**
27: **return** *n*
28: **if** *jump_3D* (*n,* $\vec{d}_z$*, s, g)* is not *null* **then**
29: **return** *n*
30: **if** *jump_3D* (*n,* $\vec{d}_{xy}$*, s, g)* is not *null* **then**
31: **return** *n*
32: **if** *jump_3D* (*n,* $\vec{d}_{xz}$*, s, g)* is not *null* **then**
33: **return** *n*
34: **if** *jump_3D* (*n,* $\vec{d}_{yz}$*, s, g)* is not *null* **then**
35: **return** *n*
36: **return** *jump_3D* (*n,* $\vec{d}$*, s, g)*

Figure 24. Pseudocode of the *jump_3D* function

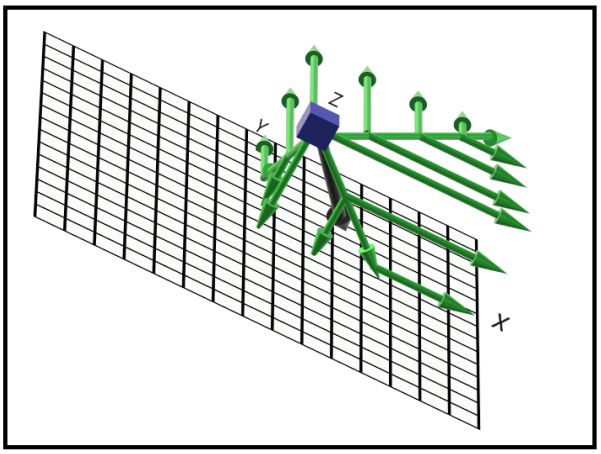

Figure 25. Illustrates the operation of the *jump_3D* function with *xyz*-diagonal direction parameter. The green colored arrows illustrate how the search expands on a 3D space.

6. IMPROVEMENTS TO JPS_3D

Based on some initial tests, JPS_3D pathfinding algorithm provides a reasonable performance compared to A* algorithm. However, the increase of the performance is not the amount as has been documented in the 2D case. Naturally, it is not even expectable to have similar performance gain. JPS and JPS_3D Algorithms are quite much different, and the 3D pathfinding environment is fundamentally different. Because the motivation of this thesis was to establish optimal solution for JPS_3D as JPS and A* algorithm offer in 2D, those optimality requirement sets constraints of how much the algorithm can be transformed. Therefore, the modification the algorithm must preserve the optimality and the online requirement.

An algorithm is a process, it takes input information, manipulates it and gives the result as an output. It is obvious that the same outcome can be reach with several different ways (algorithms). To improve the speed of the solution process, requires the identification of possible ineffective structures and operations. It is noticeable the purpose of this kind of optimization is not to change the outcome of the algorithm but only optimize the structures to improve performance. From the perspective of performance, the essential structures of an algorithm can be divided to storing and processing data. The JPS pathfinding process contains two parts, building a *graph* and *graph traversal* algorithm for solving it. In this thesis the *graph traversal* algorithm is not the target of the optimization. The optimizing is focuses on the minimization of the total time complexity of building the graph and using the graph. Even though a small graph is favorable, creating a graph requires some computational recourses. Optimization of pathfinding require balancing the computational resources between the graph creation and the graph using.

A characteristic feature of the pathfinding is that the algorithms processes a large amount of data. Even though the data manipulations might be a trivial arithmetic operations like distance comparison between the nodes, the same process is typically repeated for the large data set. Any inefficiency on repetitive structures has a cumulative effect. It is also possible that some of the gathered data are utilized inefficient way. Some intermediate

results might be discarded even though they might consist of valuable information. Therefore, efficiency of the data structures, storing and manipulation methods are important.

6.1. RECURRENCE

Pathfinding is a type of a sorting problem. As in an any sorting algorithm, a considerable number of entities are required to be compared with each other, which means that some repetition is unavoidable. From the performance and the efficiency perspective, recurrence should occur only when it is necessary. Some of the recurrence can be a part of the natural cycle of the algorithm which cannot be effectively avoided. In a pathfinding process, a certain type of environment can also cause an excessive repetitive cycle when numerous equivalently prominent paths have to be evaluated.

Algorithms use computational recourses for processing information. That information (data) can be intermediate results or the final output. When the algorithm has processed information, it can reject the result after use, or store it for later use. Rejecting the processed information is an appropriate option if the information is not required any longer. In some cases, the previously gathered information might become obsolete over the time and its storing is unnecessary. Even though the information would be valuable for later use, storing and reading of information are computational processes. To be useful for store information, it is required to be a computationally lighter/faster process than processing the same information again.

If it is assumed that the required computational recourses for processing a certain information remains the same during an entire process, storing the information for later use can be a competent method. The benefit of storing the information is obvious, if the same information is used multiple times. Additionally, previously gathered information can be an input information for other information. Reuse of the information can avoid unnecessary processing and save computational recourses. The value of the information depends how much it can save the computational recourses before it becomes obsolete. Obviously, the more computational resources are required for gathering and processing

the information, the more the reuse of the same information can save the computational resources. Even though it is obvious that also the reusing and storing information requires additional memory and computational resources, the overall benefit depend on the available recourses and priorities.

6.1.1. PRUNING RECURRENCE

It is inevitable that JPS and JPS_3D algorithms consist of some recurrence, which cannot be effectively avoided. This is especially the case when the *jump_3D* function evaluates a *forced neighbor*. Generally, the whole pruning process is recurrent when each new iteration round processes partly the same set off nodes as the previous iteration round did. This issue holds also with the original JPS algorithm, however, the problem is less significant in a 2D space when only one node on each side must be evaluated. In a 3D space computational load is increasing significantly, as every step forward require the algorithm to evaluate at least 8 nodes.

The methods for mitigating recurrence issues are limited, as the evaluation of a node require comprehensive information about other nodes around it. Approach for storing all previous evaluated information and using it later during the process is not necessarily giving a performance benefit. The problem is that the gathered information about neighbor nodes might not be valuable enough to be stored. For example, each node has a *Boolean* value to specify if the node is passable. Such kind of primitive information is always accessible and readable with the same computational effort as storing it to somewhere else. Ideally reusable information is something which has a high information value, and which is a by-product of other process so that the required additional processing is minimum.

Analyzing a different scenario of pruning reveals that there is one case when the previously processed information is valuable enough and reusable. This situation occurs when all the evaluated neighbor nodes around the node are passable. The information about non-blocked neighbor is valuable, as it implies that applying the *forced neighbors*

pruning rules is unnecessary. That information could be stored on a single Boolean value the node and its value could be then be check before applying any pruning rules.

Naturally any reusable information about the environment requires that the environment does not change during the pathfinding process or the environment is assumed to be static. Generally, any changes on the environment during a pathfinding process are significant and can cause the current pathfinding process to become outdated. Therefore, any change on the environment requires an instantaneous restart of the entire pathfinding process. This procedure is necessary for ensuring that the evaluated path exist, and the path is verified to be the shortest. If it is assumed that any changes on the environment requires the restart of the entire pathfinding process, one can also assume that all the previously processed information is valid as long the environment remains static. This assumption gives a possibility to use a previously processed information during a pathfinding process.

6.1.2. PRUNING OPTIMIZATION

The pruning function evaluates the eight nearest neighbors during one iteration round; however, there are usually large amount of consecutive iteration rounds in each direction. When three consecutive iterations are conducted in any single axis direction, the pruning function has evaluated the cubic shape space size of 3*3 nodes, around the evaluated nodes. Let us assume that the jump function is called three times for evaluating nodes: n_1, n_2, n_3 (Figure 26) If all those iterations will return the information about the empty space, then conducting the pruning to the same nodes from the opposite direction (n_3, n_2, n_1) will also return the same result. During that pruning sequence, the node n_2 is located between the nodes n_1 and n_3, so previously acquired information can be used to indicate that neighbor nodes n_1 and n_3 has no nearest neighbors which are occupied by obstacles. Therefore, every separate pruning process in directions of *x*, *y* or *z*-axis would always return the same result about node n_2 and further pruning would be unnecessary.

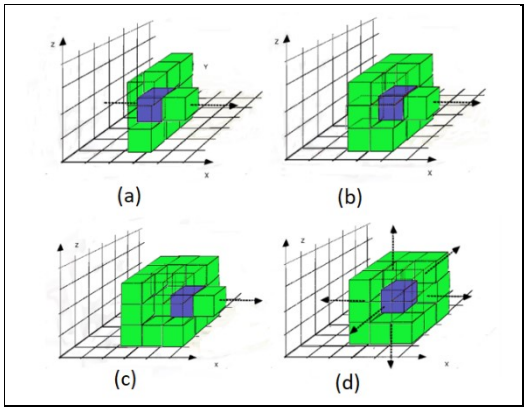

Figure 26. Pruning three consecutive nodes along the x-axis. The algorithm must evaluate the green nodes. In case all the nodes are non-blocked the blue node on the center (d) is possible to tag with "free" information so that the algorithm knows that further evaluation around the blue node is not needed regardless from the pruning direction.

Storing information about the open space, around some specific node, requires theoretically one *bit*. That information can be read, before the pruning function is applied on the node. The benefit of this is significant when all three coordinate-axis with positive or negative direction vector could use the same information for preventing unnecessary pruning. The overall improvement for the performance depends on how much overlapped pruning is happening. A more complex environment generates more *jump points* and therefore improvement should be more significant.

6.1.3. PRUNING OPTIMIZED JUMP_3D

This feature requires a moderate modification on the current algorithm (see Figure 28). Some new notations are added to define the algorithm. To indicate that the (3*3*3) cubic grid around the node n does not contain any obstacles, n is defined to be "*free*". The direction of pruning is defined by $\vec{d}_p$ and it contains six individual Boolean values $\{\vec{d}_{+x}, \vec{d}_{-x}, \vec{d}_{+y}, \vec{d}_{-y}, \vec{d}_{+z}, \vec{d}_{-z}\}$. The Boolean value represents all the main coordinate axes with positive or negative directions. Additionally, the notation (n_{-2}, n_{-1}, n) refers to node n and two "previous" nodes. The term "previous" is relative, and it depends on the direction of pruning $(\vec{d}_p)$ of the node n, (see Figure 27).

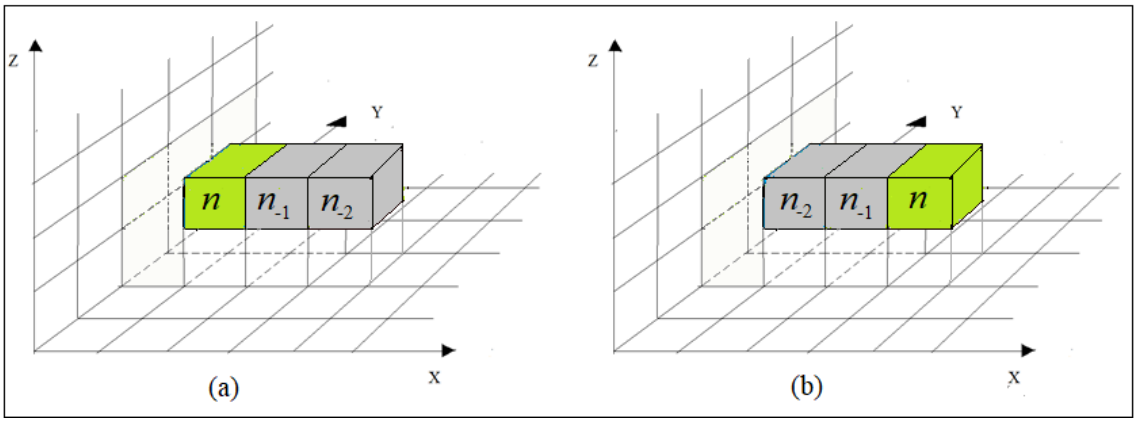

Figure 27. Figure (a) shows node *n* (green) and two "previous" nodes (gray) when the pruning direction $\vec{d}_{-x}$ is true. Figure (b) shows node *n* (green) and two "previous" nodes (gray) when the pruning direction $\vec{d}_{+x}$ is true.

The actual implementation requires a mechanism which can detect if further pruning is unnecessary (line 6). The algorithm sets up the Boolean value if no *forced neighbors* are detected. It is noticeable that each of the sixth direct pruning directions requires its own Boolean value $\vec{d}_p$ (line 10). Additionally, the algorithm requires a mechanism which can detect if the node has at least three adjacent consecutive neighbors (n_{-2}, n_{-1}, n) which have no *forced neighbors* (lines: 40…42). In that case, the algorithm sets the Boolean value to the node n_{-1} that it is *free*. The attribute *free* indicates that pruning is not needed.

Input: *x*: initial node, $\vec{d}$: *direction*, *s*: *start*, *g*: *goal*

1. : $n \longleftarrow step(x, \vec{d})$
2. : **if** *n* is an obstacle or out of grid **then**
3. : **return** null
4. : **if** *n* = *g* then
5. : **return** *n*
6. : **if not** *n* is *free* **then**
7. : **If** ($\vec{d} == \vec{d}_{\pm x}$, **or** $\vec{d} == \vec{d}_{\pm y}$, **or** $\vec{d} == \vec{d}_{\pm z}$,)
8. : **set** $\vec{d}_p$ of node *n* to $\vec{d}$
9. : **if** *n*' ∈ neighbors(*n*) s.t. *n*' is *forced* then
10. : **return** *n*
11. : **if** $\vec{d}$ is diagonal on *xy*-plane **then**
12. : **if** *jump_3D (n,* $\vec{d}_x$*, s, g)* is not *null*) **then**
13. : **return** *n*
14. : **if** *jump_3D* (*n,* $\vec{d}_y$*, s, g)* is not *null* **then**
15. : **return** *n*
16. : **if** $\vec{d}$ is diagonal on *xz*-plane **then**
17. : **if** *jump_3D* (*n,* $\vec{d}_x$*, s, g)* is not *null* **then**
18. : **return** *n*
19. : **if** *jump_3D* (*n,* $\vec{d}_z$*, s, g)* is not *null* **then**
20. : **return** *n*
21. : **if** $\vec{d}$ is diagonal on *yz*-plane **then**
22. **:** **if** *jump_3D* (*n,* $\vec{d}_y$*, s, g)* is not *null* **then**
23. : **return** *n*
24. : **if** *jump_3D* (*n,* $\vec{d}_z$*, s, g)* is not *null* **then**
25. : **return** *n*
26. : **if** $\vec{d}$ is diagonal on *xyz*-space **then**
27. : **if** *jump_3D* (*n,* $\vec{d}_x$*, s, g)* is not *null* **then**
28. : **return** *n*
29. : **if** *jump_3D* (*n,* $\vec{d}_y$*, s, g)* is not *null* **then**
30. : **return** *n*
31. : **if** *jump_3D* (*n,* $\vec{d}_z$*, s, g)* is not *null* **then**
32. : **return** *n*
33. : **if** *jump_3D* (*n,* $\vec{d}_{xy}$*, s, g)* is not *null* **then**
34. : **return** *n*
35. *:* **if** *jump_3D* (*n,* $\vec{d}_{xz}$*, s, g)* is not *null* **then**
36. : **return** *n*
37. *:* **if** *jump_3D* (*n,* $\vec{d}_{yz}$*, s, g)* is not *null* **then**
38. : **return** *n*
39. : **if** n has no *forced neighbor* on direction $\vec{d}_p$ **then**
40. **:** **if** n_{-1} has no *forced neighbors* on direction $\vec{d}_p$ **then**
41. **:** **if** n_{-2} has no *forced neighbors* on direction $\vec{d}_p$ **then**
42. **:** **Set** n_{-1} is *free*
43. **: return** *jump_3D* (*n,* $\vec{d}$*, s, g)*

Figure 28. Pruning optimized *jump_ 3D* function

6.1.4. JUMP RECURRENCE

When the *JPS* algorithm is processing nodes, it usually has several other *jump points* on the *successor* list. These nodes are on virtually arbitrary locations from each other. Additionally, all the *jump point* locations are the starting points for the new searches. It is unavoidable that in many cases *jump_3D* must process some nodes that are already evaluated by another *jump_3D* function call (see Figure 29). This is a typical phenomenon and necessary for ensuring the optimality. The Reason for the repetition is that the direction is an essential parameter how the nodes are evaluated. The same node can be approached from eight different directions in a two-dimensional space. On a three-dimensional space there are 26 different directions to approach a single node. From the perspective of JPS algorithm they are separate cases and require to be evaluated separately. Additionally, there are also cases where the same node is evaluated multiple times from the same trajectory. Those situations are also individual cases because of the distance variations.

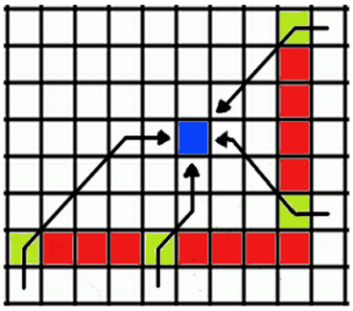

Figure 29. The green nodes represent the *jump points*. The black arrows are illustrating how the blue node can be reached from the several various directions and locations on a two-dimensional space.

6.1.5. OPTIMIZATION OF THE RECURSION

One of the most fundamental parts of the original JPS algorithm is the definition of the *jump point* search. The original JPS algorithm defines that a search of a *jump point* continues until the *jump point* is found, or it will return *null* if the search will end on a wall or outer bounds. The *jump point* is strictly defined without exceptions so that only the smallest set of potential turning points are returned, which is an effective way to keep the size of the *graph* minimal. For instance, if the space is empty nothing except the *goal* node is returned to the *graph traversal* algorithm.

However, the original definition of the *jump point* has some drawbacks especially on a 3D space where the search space is typically larger. For instance, a single call of *jump_3D* with a *xyz*-diagonal direction parameter will expand to a 3D space scanning. It means that the single *jump_3D* call can trigger the evaluation of a great number of nodes. The outcome is that considerable number of recursive *jump_3D* calls will be generated which is time consuming. Additionally, many of the *jump_3D* calls will also travel to a "wrong direction". Unfortunately, it is impossible to judge what is a "wrong direction" until the whole path has been found. Therefore, it is not possible to ignore any search direction.

When *jump_3D* has no mechanism to interrupt the chain of recursions. This is problematic in a large space as it cannot change its direction. Also, the *graph traversal* algorithm with its heuristics cannot effectively guide the search towards the goal. One possible workaround for mitigating this issue is limiting the depth of the consecutive recursive calls. It operates like an interruption mechanism to the *jump_3D* function so that the *graph traversal* algorithm can frequently select the most prominent nodes. A modification requires extending the original *jump point* definition so that also a certain distance from a parent node location satisfies the definition of a *jump point*. For instance, setting up recursion the depth limit to 3, *jump_3D* returns the third node regardless if it fulfills other part of definitions for the *jump point*. Naturally, before the *recursion depth limit* reaches the maximum value, the algorithm might return a *jump point* based on the original JPS definition. However, the *identify_successor_3D* function will process those distance-based *jump points* the same way as any other *jump points*.

Optimality of the path is an absolute requirement and, therefore, extending the definition of the *jump points* raises a question about possible side effects. It is obvious that a *jump point* cannot be a randomly selected location. The original algorithm generates a minimum set of *jump point* points which are required for evaluating an optimal path. However, that minimum set is typically significantly larger than the set of nodes which actually belongs to the optimal path. The original algorithm does not guarantee that each of the evaluated *jump point* will necessary belong to the optimal path. Depending on the environment the *jump* function can generate a significant amount of "non-relevant" *jump points*. The *jump* function is necessary to generate additional *jump points* because the validity of an individual *jump point* is possible to verify only after the complete path has been found. Despite the "non-relevant *jump points* the *graph traversal* algorithm can eventually determine the optimal path.

Even though if the *jump point* does not belong to the optimal path, each of the evaluated *jump points* always represents a local optimal path from *p(x)* via *x* to its *natural neighbors*. That definition comes from the original JPS algorithm. To extend the original definition supporting the "distance-based" *jump point* one has to analyze the original search process. In a two-dimensional space, searching *jump points* can be done by in a direct (horizontally/vertically) or diagonally. The direct moving is obviously always an optimal straight path from the parent node to *node* and to its *natural neighbor*. Moreover, any arbitrary node on the same straight trajectory would also form an optimal path from the parent node to that node.

The diagonal moving direction can also generate a broken line from *p(x)* via *x* to its *natural neighbors* (Figure 30). Those curved sub-paths have two sections so that path π_1 from *p(x) to x* to is always optimal regardless the distance between *p(x)* and *x*. The other section is a path from node *x* to any its *natural neighbors* which is also optimal. However, it is obvious that the optimality is not limited only to its *natural neighbors*. Any arbitrary node n_i on that same trajectory forms an optimal sub-path π_2 from *x* to n_i. Combining paths π_1 and π_2 together, forms the path from *p(x)* via *x* to n_j which is always optimal regardless length of the components.

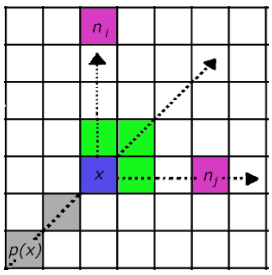

Figure 30. Illustrates the diagonal *jump* function call on two-dimensional space where the path from p*(x)* to *x* (the blue node) is allays optimal to its *natural neighbors* (the green nodes), additionally the paths from x to n_i and x to n_j always forms optimal paths. Therefore, the combination of the paths ($p(x)$, x, n_i) and ($p(x)$, x, n_j) always forms optimal path regardless their lengths.

In summary, a distance-based *jump point* is not necessarily a location where the path is changing its direction, as it is more like a temporarily stored location for the *graph traversal* algorithm. This new type of *jump point* stores intermediate result from the *jump_3D* function. That location will be the new starting point for the new recursive function calls. The prominence of the *jump point* is immediately evaluated by a *graph traversal* algorithm. Therefore, the limited recursion depth with the *graph traversal* algorithm makes the search process more responsive; the algorithm can search in the prominent direction and reduce unnecessary deep search on a non-prominent direction.

It is noticeable that this method does not require any major changes to the *jump_3D* function. Also, the original definition for the *jump point* is still valid and the original *jump points* are evaluated as previously. Each of the new *jump point* will have valid parent and direction parameters and the *search order* stays the same. Compatibility with the original algorithm principles allows that the *jump_3D* function with limited recursion depth preserves the optimality of a path.

6.1.6. RECURSION OPTIMIZED JUMP_3D

This feature is relatively straightforward to implement requiring only a few additional counters and *condition statements* to the current *jump_3D* function. The depth of the recursion is defined with one additional counter, and it decrements on every recursive call until the counter will reach zero. The pseudocode of revised *jump_3D* is showed in Figure 31. One of the main features is the (recursion) *depth*-counter on the line 1. It decreases during each call of *jump_3D* function. The detection of recursion depth is conducted on each *jump_3D* statements. The value of the *depth counter* is checked on lines (10, 12, 15, 17, 20, 22, 25, 27, 29, 31, 33, 35), and if it is zero the algorithm must return the current evaluated node *n.*

Input: x initial node, $\vec{d}$: *direction*, s: *start*, g: *goal*, *depth*

```
1.  :  depth = depth-1
2.  :  n ⟵ step(x, d⃗)
3.  :  if n is an obstacle or out of grid then
4.  :       return  null
5.  :  if n = g then
6.  :       return  n
7.  :  if n' ∈ neighbours(n) s.t. n' is forced then
8.  :       return  n
9.  :  if d⃗ is diagonal on xy-plane then
10. :       if  jump_3D ((n, d⃗_x, s, g) is not null or depth <= 0) then
11. :                                   return n
12. :       if  jump_3D ((n, d⃗_y, s, g) is not null or depth <= 0) then
13. :                                   return n
14. :  if  d⃗ is diagonal on xz-plane then
15. :       if  jump_3D ((n, d⃗_x, s, g) is not null or depth <= 0) then
16. :                                   return n
17. :       if  jump_3D ((n, d⃗_z, s, g) is not null or depth <= 0) then
18. :                                   return n
19. :  if  d⃗ is diagonal on yz-plane then
20. :       if  jump_3D ((n, d⃗_y, s, g) is not null or depth <= 0) then
21. :                                   return n
22. :       if  jump_3D ((n, d⃗_z, s, g) is not null or depth <= 0) then
23. :                                   return n
24. :  if  d⃗ is diagonal on xyz-space then
25. :       if  jump_3D ((n, d⃗_x, s, g) is not null or depth <= 0) then
26. :                                   return n
27. :       if  jump_3D ((n, d⃗_y, s, g) is not null or depth <= 0) then
28. :                                   return n
29. :       if  jump_3D ((n, d⃗_z, s, g) is not null or depth <= 0) then
30. :                                   return n
31. :       if  jump_3D ((n, d⃗_xy, s, g) is not null or depth <= 0) then
32. :                                   return n
33. :       if  jump_3D ((n, d⃗_xz, s, g) is not null or depth <= 0) then
34. :                                   return n
35. :       if  jump_3D ((n, d⃗_yz, s, g) is not null or depth <= 0) then
36. :                                   return n
37. : return jump_3D (n, d⃗, s, g)
```

Figure 31. Pseudocode of the recursion optimized *jump_3D* function

6.2. COMBINED OPTIMIZATION

The pruning optimization and the recursion depth optimization utilizes different methods to improve efficiency. Therefore, it is possible that JPS_3D algorithm can utilize both optimization methods simultaneously. Implementation is relatively straightforward to do by merging the optimized algorithms together.

7. PERFORMANCE COMPARISONS

The motivation for conducting performance tests is to find out if the JPS_3D algorithm and its optimized versions have a competitive performance. The performance is defined by a *speed ratio* which is calculated by dividing a *search time* of reference algorithm by a *search time* of JPS_3D. In this case A* algorithm was chosen to be the reference algorithm. Both algorithms offer equivalent optimality and both algorithms use about the same amount of memory recourses. Other important result of testing is to validate the optimality of a path. Even though the 3D adaptation of the JPS was designed and implemented by using the theoretical principles of JPS, the test results are essential to prove that the new version truly provide an optimal path.

7.1. TEST ARRANGEMENTS

The tests were conducted using a 3D A* algorithm as the reference algorithm. The tested algorithms were JPS_3D and its optimized variations which were also tested separately against the 3D A* algorithm. The pruning optimized JPS_3D algorithm is called by JPS_3D_P. The recursion depth optimized JPS_3D algorithm is called JPS_3D_D and JPS_3D_PD algorithm utilizes both the pruning and the recursion depth optimization. The test environments were consisted of varieties of sizes and shapes including a large and complex spaces which require considerable amount of computation and processing time. The tests also included an artificial model of a ten-story building.

Each test round stores information about the test results like a search time, a length of the path, the differences in elevation between *start* and *goal,* and number of processed nodes. Additionally, each test result saves general information about the test setup itself like utilized algorithms, total volume of the space, the fill rate of the space, and the name of the test environment.

Implementation of the algorithms and the tests were done by *C#* language using *Unity 3D* engine. The operating system was Windows 10 with up to date operating system updates.

Unfortunately, *Windows 10* platform with *C#* program code is not the ideal platform for accurate and repeatable tests as the platform has numerous background processes and memory management which the user cannot control. To mitigate those issues, the tests were repeated many times for each problem instance.

The test program is fully automated which loaded different 3D models of space and obstacles. The test program takes the input parameters for selecting the utilized algorithm, it selects randomly the *start* and *goal* locations, generates a certain number of obstacles, and conducts selected the number of the tests. At the end, the test results were written on *csv* files which were then imported to *excel* sheets for further analysis.

7.2. PATHFINDING ON AN OPEN SPACE

The test arrangement consists of an empty space with randomly selected *start* and *goal* locations. The size of the space contains ~800000 traversable nodes. The test case was repeated 230 times. The test results revealed that A* algorithm outperformed JPS_3D, JPS_3D_P, JPS_D and JPS_3D_PD algorithms. It is noticeable that the *recursion depth* limit for the JPS_3D_P was set to 1. The open space test is particularly challenging for the JPS_3D algorithm without any optimization. Without any obstacles on the space, JPS_3D algorithm has to scan the entire space. For the A* algorithm, the open space test is computationally light as the heuristic function can accurately guide the search.

Figure 32. shows the speed ratio of (JPS_3D/A*) where A* algorithm is much faster than JPS_3D. Average speed ratio of JPS_3D is only 0.00189. However, it is noticeable that the longer the path the better the speed of JPS_3D algorithm become.

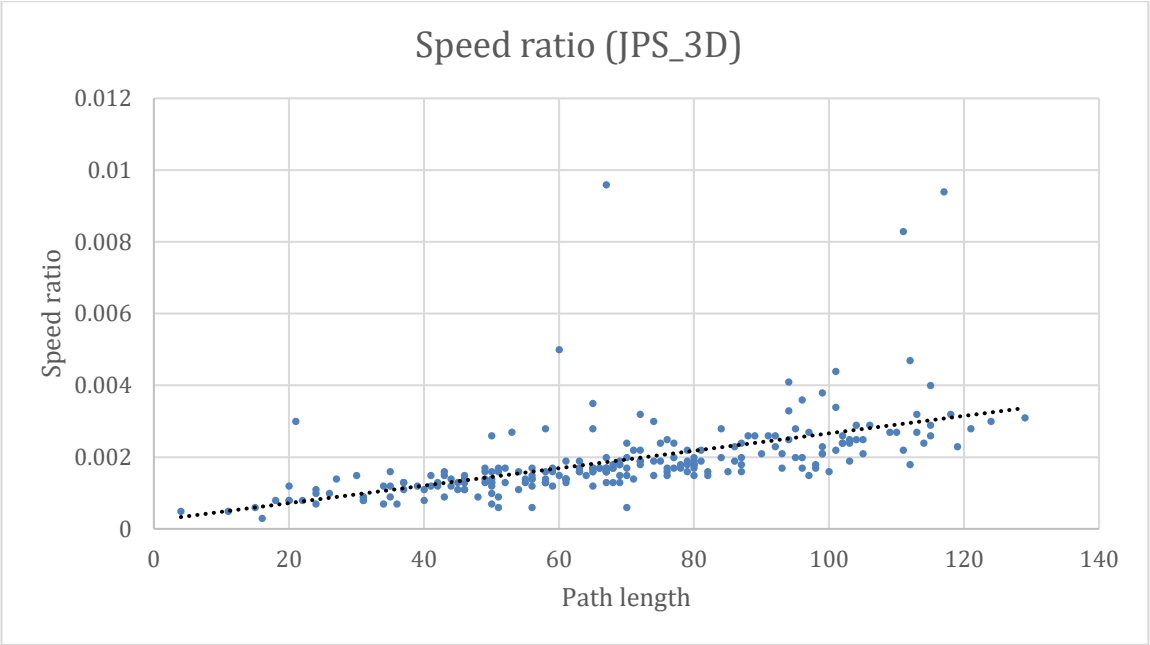

Figure 32. Comparison of the running time of JPS_3D vs. A* algorithm as function of the path length.

Using the pruning optimization does not improve the speed of JPS_3D_P algorithm (Figure 33). On a totally empty space without overlapping *jump* function calls the pruning optimization does not have any advantage. The average speed factor is only 0.00181. However, it is noticeable that the longer path length improves the speed of JPS_3D_P algorithm.

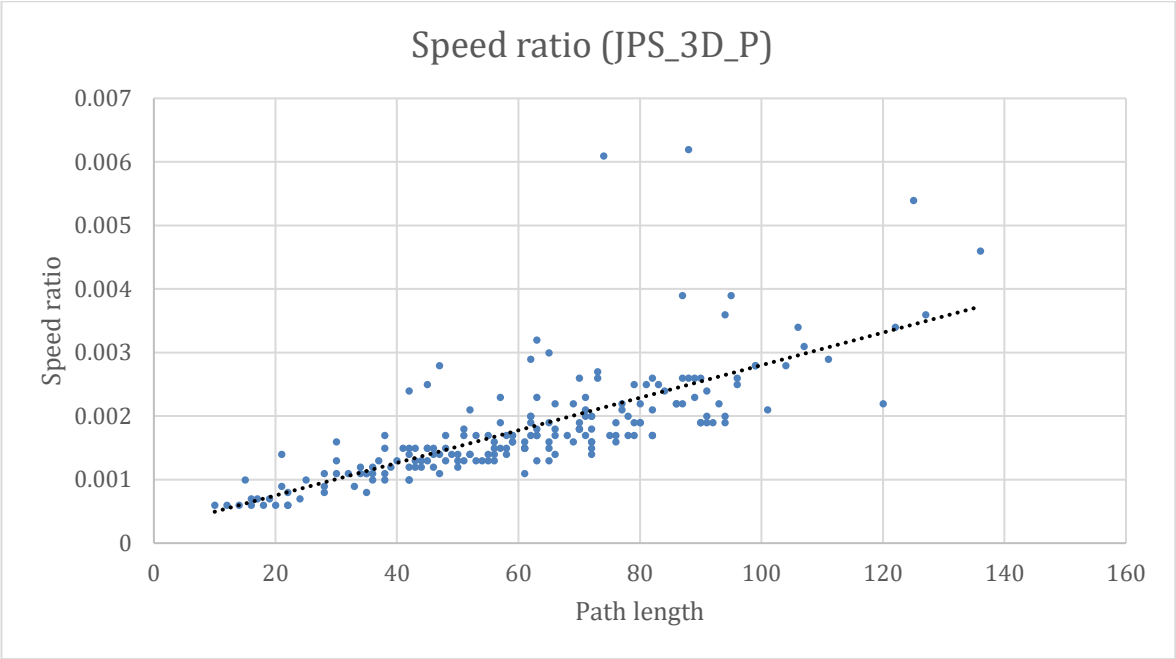

Figure 33. Comparison of the running time of JPS_3D_P vs. A* algorithm as function of the path length.

Using the recursion depth optimization improves the speed of JPS_3D_D (Figure 34). Even on a totally empty space the recursion depth optimization can drastically improve the efficiency compared with non-optimized JPS algorithm. The improvement is possible when unlimited and usually a long recursive *jump* function calls are terminated using the recursion depth limit. The average speed factor is 0.190684. The speed improvement is about 100 times compared with JPS_3D and JPS_3D_P.

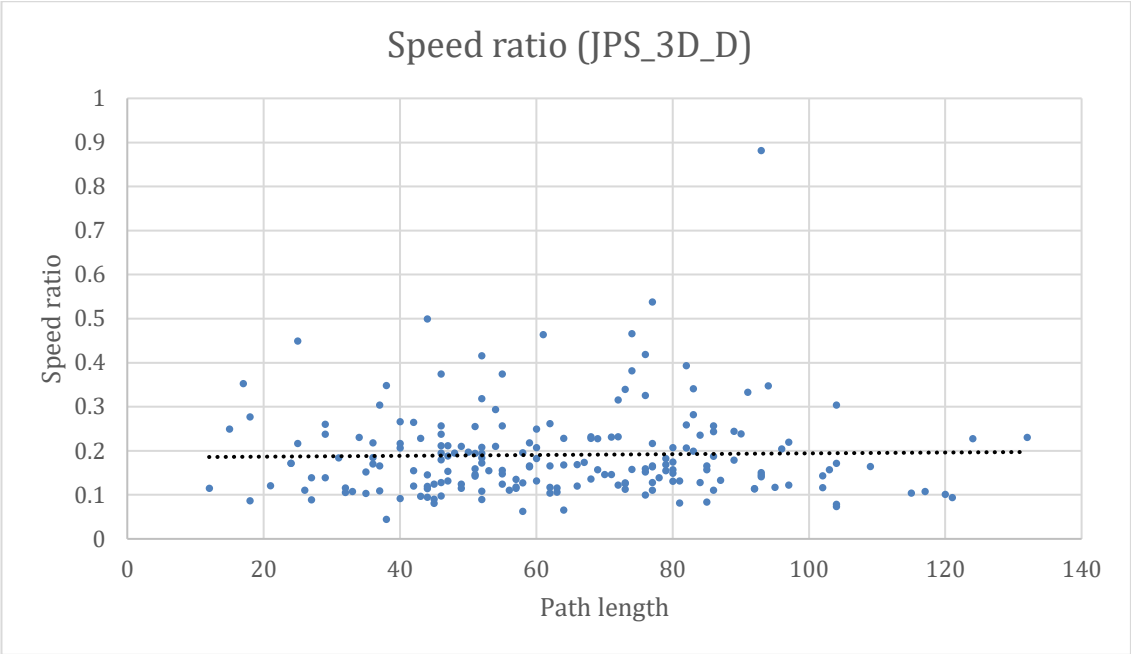

Figure 34. Comparison of the running time of JPS_3D_D vs. A* algorithm as function of the path length.

Using the recursion depth optimization with the pruning optimization improves the speed of JPS_3D_PD (Figure 35). On the completely empty space, the recursion depth optimization improves the performance drastically by preventing unnecessary long recursion chain. Based on the previous tests results, the *pruning* optimization does not have big effect on empty space. The average *speed factor* for JPS_3D_PD is 0.208843.

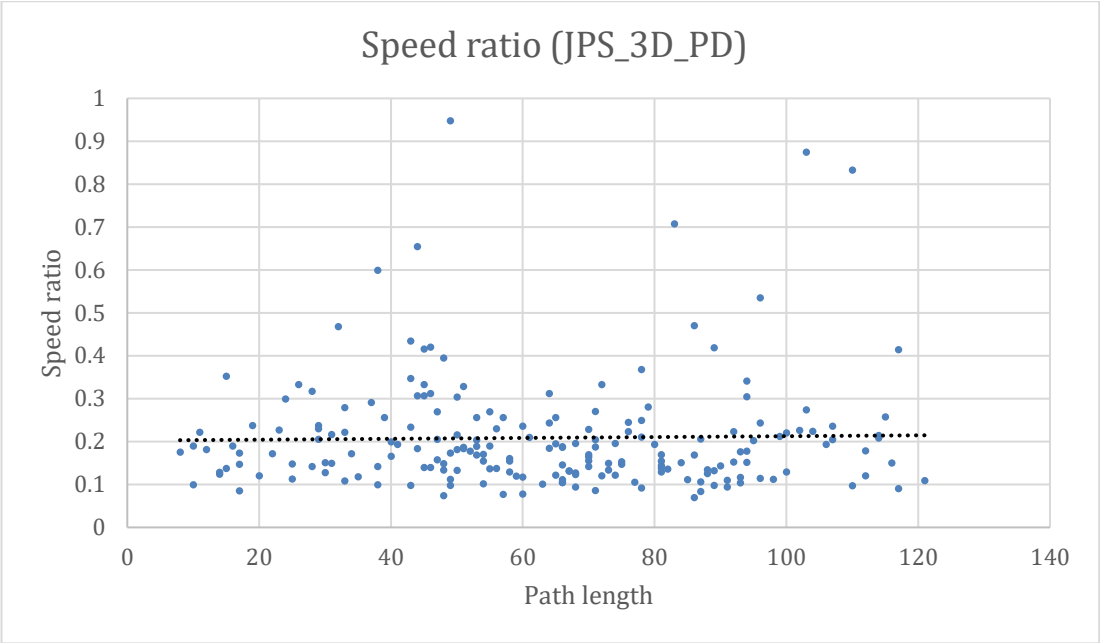

Figure 35. Comparison of the running time of JPS_3D_PD vs. A* algorithm as function of the path length.

7.3. PATHFINDING INSIDE RANDOMLY FILLED SPACE

The purpose of the test is to recognize how the different fill rates effect on the speed of the JPS_3D algorithm and its optimized variants. In this case, the space contains total ~45000 traversable nodes. The tests are conducted so that each test round will generate a certain number of randomly located obstacles on the test space. The test cases are repeated 50 times with the same fill rate. The fill rate starts from 0% and it is increased by 5% steps until the maximum fill rate 80% is reached. Higher fill rates are not measured due the decreasing probability of the traversable path. The measurements are averaged and shown on below.

The test results show that there is a nonlinear correlation between the speed ratio and the fill rate. The test result (Figure 36) reveals that under the 5% fill rate JPS_3D is uncompetitive. Increasing the fill rate up to 5% increase the performance to an equal level with the A* algorithm. Increasing the fill rate up to 20 % increases the speed ratio up to ~1,5. which seem to be local maximum. The fill rate between 25% and 35% seems to give

reasonable ~1.45 speed ratio. With the fill rates between 20% and 65% the average speed ratio drops linearly down to ~1. By increasing the fill rate up to 75% increases the speed ratio up to ~1.3. The maximum fill rate 80% seems to improve speed ratio up to 2.4.

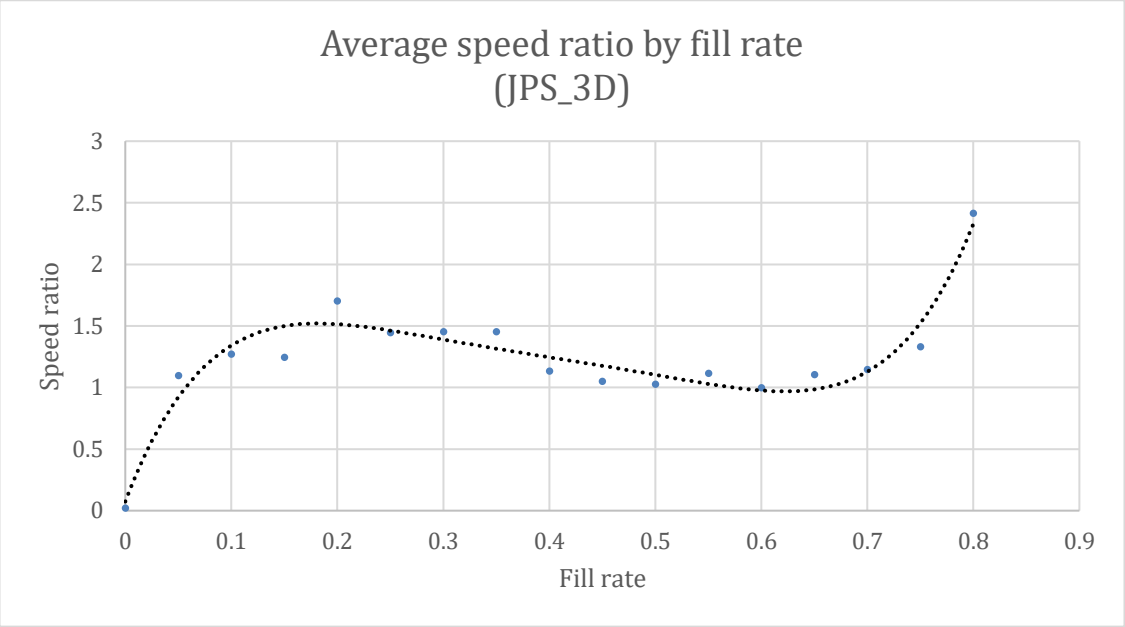

Figure 36. JPS_3D algorithm average speed ratio by function of fill rate, for random environments of obstacles.

The test result (Figure 37) shows that under the 5% fill rate JPS_3D_P is uncompetitive. Increasing the fill rate up to 5% increase the performance equal level with A* algorithm. Increasing the fill rate up to 15 % increases the speed ratio up to ~1.45. which seem to be local maximum. The fill rate between 25% and 35% seems to give reasonable ~1.45 speed ratio. With fill rates between 40% and 60 % the average speed ratio drops down to ~1.1. increasing the fill rate up to 75% increases the speed ratio up to ~1.3. Using the maximum fill rate 80% seems to improve the speed ratio drastically up to 2.4.

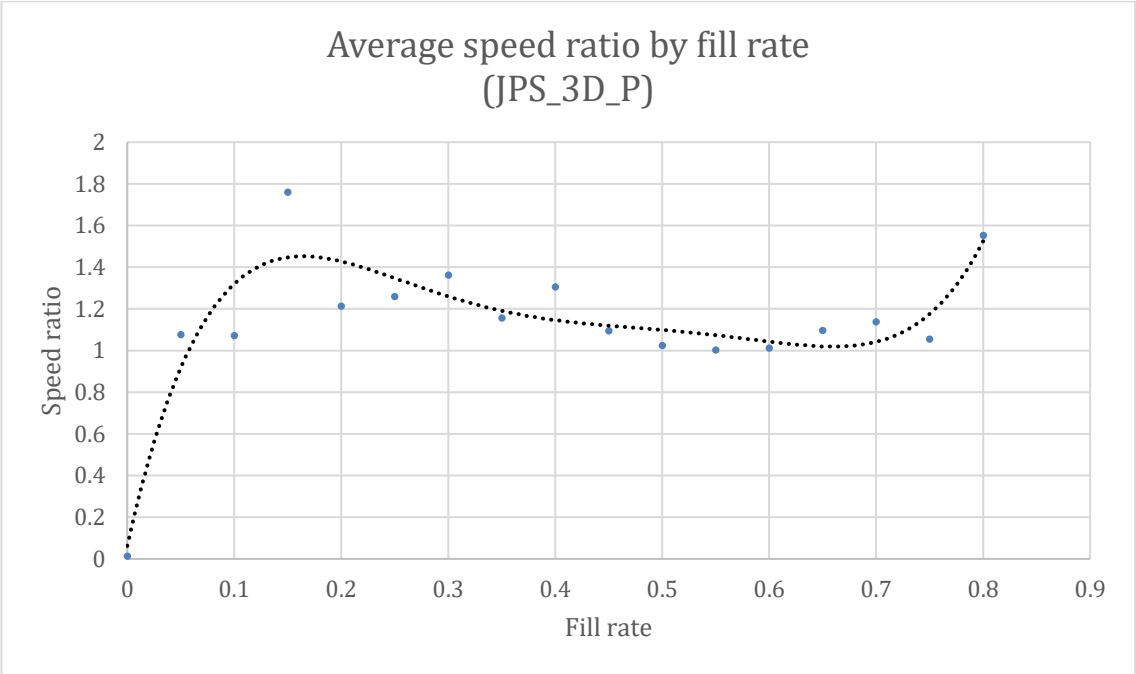

Figure 37. JPS_3D_P algorithm average speed ratio by function of fill rate, for random environments of obstacles.

The test result (Figure 38) shows that under the 5% fill rate JPS_3D_D is uncompetitive. Increasing the fill rate up to 5% increase the performance equal level with A* algorithm. Increasing the fill rate up to 15 % increases the speed ratio up to ~1,5. which seem to be local maximum. The fill rate between 40% and 75% seems to give reasonable ~1.2 speed ratio. With the fill rates between 40% and 60 % the average speed ratio drops down to ~1.1. increasing the fill rate up to 75% increases the speed ratio up to ~1.3. Using the maximum fill rate 80% seems to improve the speed ratio up to 2.4.

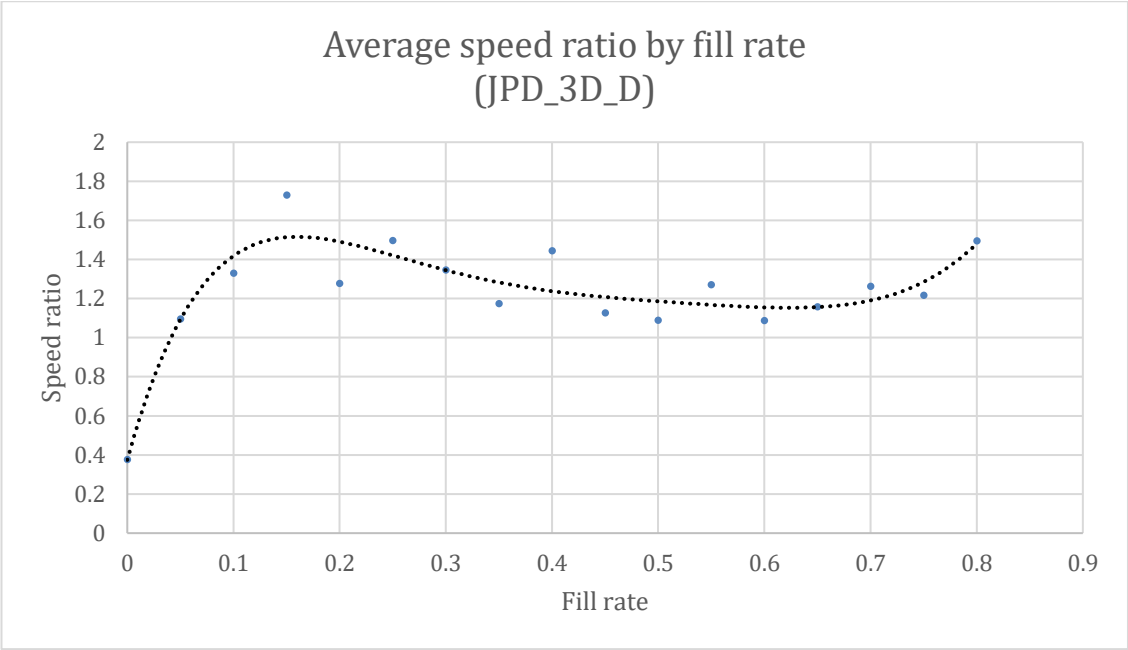

Figure 38. JPS_3D_D algorithm average speed ratio by function of fill rate, for random environments of obstacles.

The test result (Figure 39) shows that under the 7.5% fill rate JPS_3D_PD is uncompetitive. Increasing the fill rate up to 7.5% increase the performance equal level with A* algorithm. Increasing the fill rate up to 20 % increases the speed ratio up to ~1.4. which seem to be local maximum. With the fill rates between 20% and 60 %, the average speed ratio drops down to ~1.1. Increasing the fill rate over 60% significantly increased the speed ratio. With the maximum fill rate 80% seems to improve the speed ratio up to 1.8.

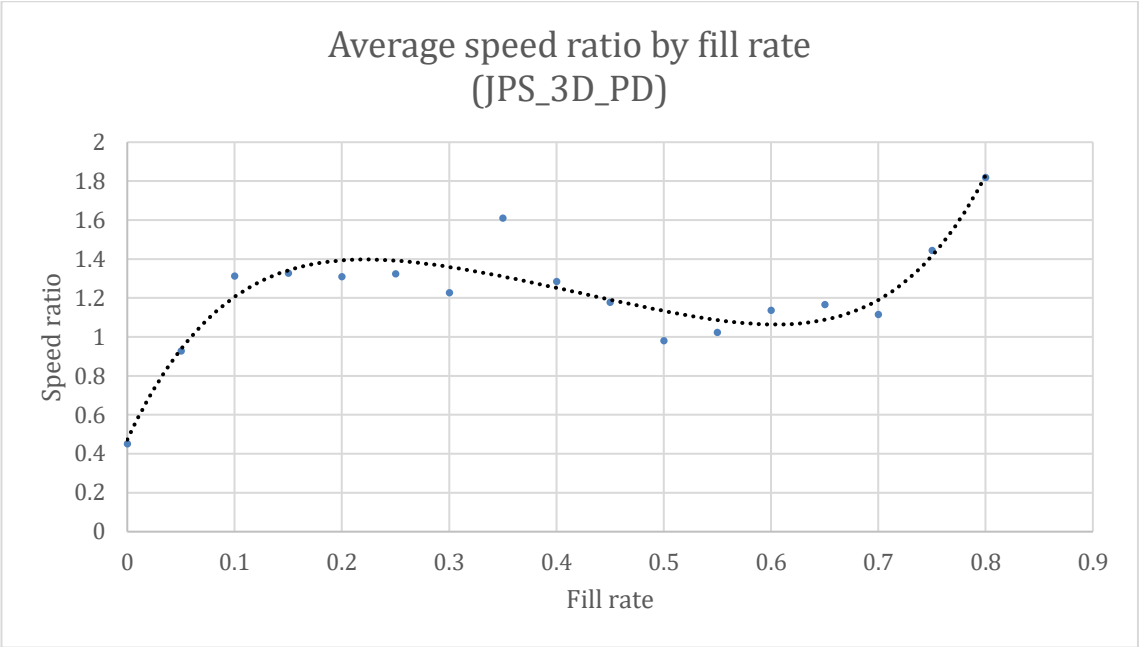

Figure 39. JPS_3D_PD algorithm average speed ratio by function of fill rate, for random environments of obstacles.

In summary, inside randomly filled space A*algorithm is not dominant as on the totally empty space. JPS based algorithms are getting more competent at randomly filled space due the obstacles. The obstacles create *jump points* and therefore, *jump* function is not needed to scan the whole space. Moreover A* algorithm loses its advantage of using heuristic function when the optimal path is getting non-straight. Additionally, longer path increases the size of *openlis*t on A* algorithm which has negative impact on the performance.

7.4. PATHFINDING INSIDE A BUILDING

This test consists of an artificial ten-story building. The building is not an exact model of any real building; however, it includes multiple pathways between the rooms, stairways, corridors and dead ends. Therefore, this model represents adequately well a 3D real-life pathfinding problem. The model contains total ~550000 traversable nodes. The tests are conducted by selecting random locations for the *start* and *goal* points, the same procedure is repeated for each algorithm and the test results are saved to *csv* (comma separated

values) file. The measurement data are presented by *xy* scatter plots so that each measurement has the *speed ratio* value presented on y-axis and the path length is presented on x-axis. The measurements are widely scattered on x-axis which indicates that there is no absolute correlation between the path length and the speed ratio. The fifth degree polynomial *trendline* is selected for illustrating the correlation between the average speed ratio and path length.

JPS_3D algorithm (Figure 40) is clearly slower than A* algorithm when the path length is less than 35. At the shorter distances, A* algorithm is generally faster than JPS_3D because a path is often a direct line. The speed ratio of JPS_3D algorithm reaches the equal level with A* algorithm when path length reaches value 35. In this test arrangement, the maximum speed ratio ~1.7 is reached when the path length is about 100. From the path length of 200 till the maximum value 500, the speed ratio stays relatively steadily on ~1.5.

JPS_3D_P algorithm (Figure 41) is also slower than A* algorithm when the path length is less than 30. At the shorter distances A* algorithm is faster because a path is often direct line. The average speed ratio of JPS_3D_P algorithm reaches the equal level with A* algorithm when path length reaches value 30. In this test arrangement, the maximum speed ratio ~2.9 is reached when the path length is about 110. From the path length of 200 till the maximum path length (~500) the speed ratio stays relatively steadily on 2.5.

JPS_3D_D algorithm (Figure 42) is also slower than A* algorithm when the path length is less than 20. At the shorter distances A* algorithm is again outperforming JPS_3D_D because a path is often direct line. However, the recursion optimization seems to have a positive effect on performance. JPS_3D_D algorithm reaches its maximum speed ratio ~3,4 when the path length is about 100. After the maximum, the speed ratio decreases and has a local minimum 2,5 when the path length reaches 280. With the longer path length up to maximum path length, the speed ratio is slightly better 2.8.

JPS_3D_PD algorithm (Figure 43) is also slower than A* algorithm when the path length is less than 20. At the shorter distances A*algorithm is outperforming because a path is often direct line. However, the pruning and the recursion optimization seems to have a

positive effect on performance. JPS_3D_PD algorithm reaches its maximum speed ratio ~4.3 when the path length is about 130. After the maximum, the speed ratio decreases and has a local minimum 3.7 when the path length reaches 300. With longer path lengths up to the maximum path length, the speed ratio is slightly better ~4.

In summary, the test results indicate that inside a building, the JPS based pathfinding algorithms seems to have generally better performance than A* algorithm can provide. Also, the optimization methods are significantly improving the results. The results are indicting that the *jump point search* is a relatively effective method when environment consists of separate spaces (rooms) and the path often travels backwards. Such kind of path is challenging for A* algorithm. The issue is that the A* algorithm utilizes heuristic function which priorities the search direction only towards the goal. In case there is a large space (dead-end) which eventually cannot offer optimal path A* algorithm requires evaluate the whole space ahead before the search can go backward.

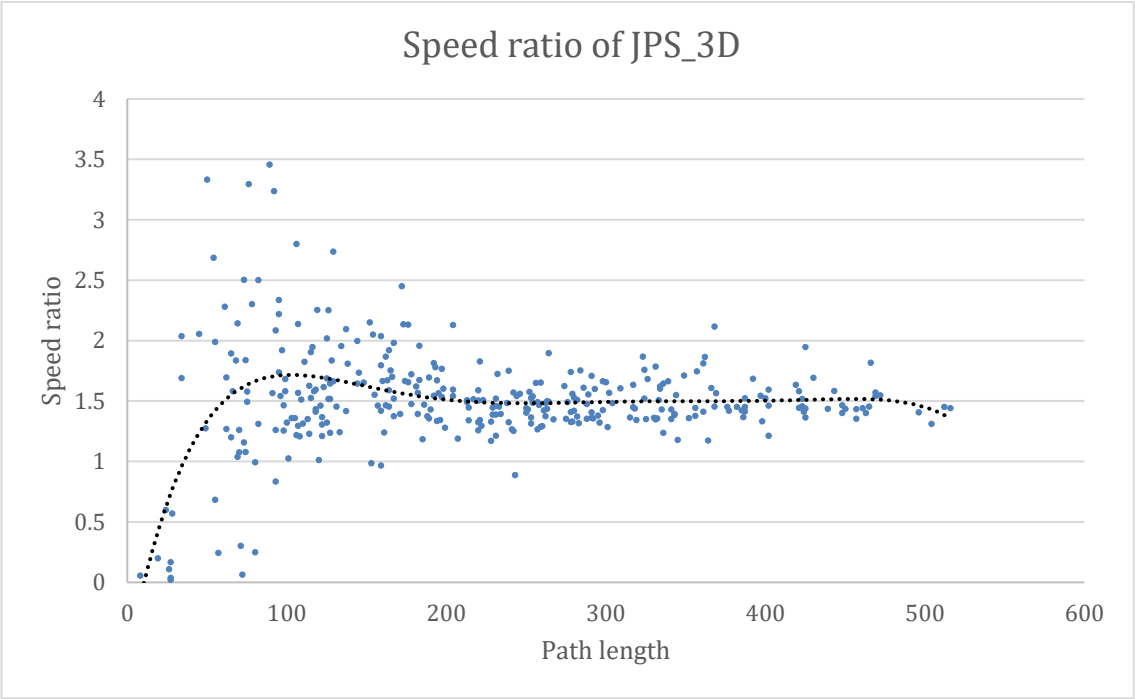

Figure 40. Speed ratio of the JPS_3D.

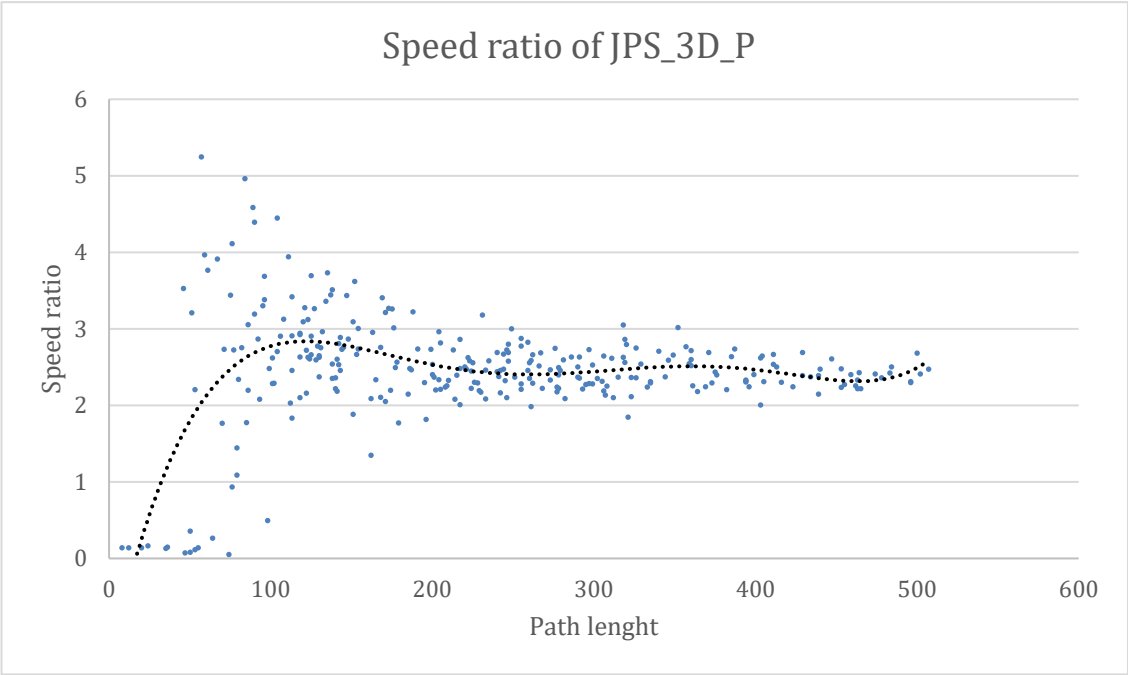

Figure 41. Speed ratio of the JPS_3D_P.

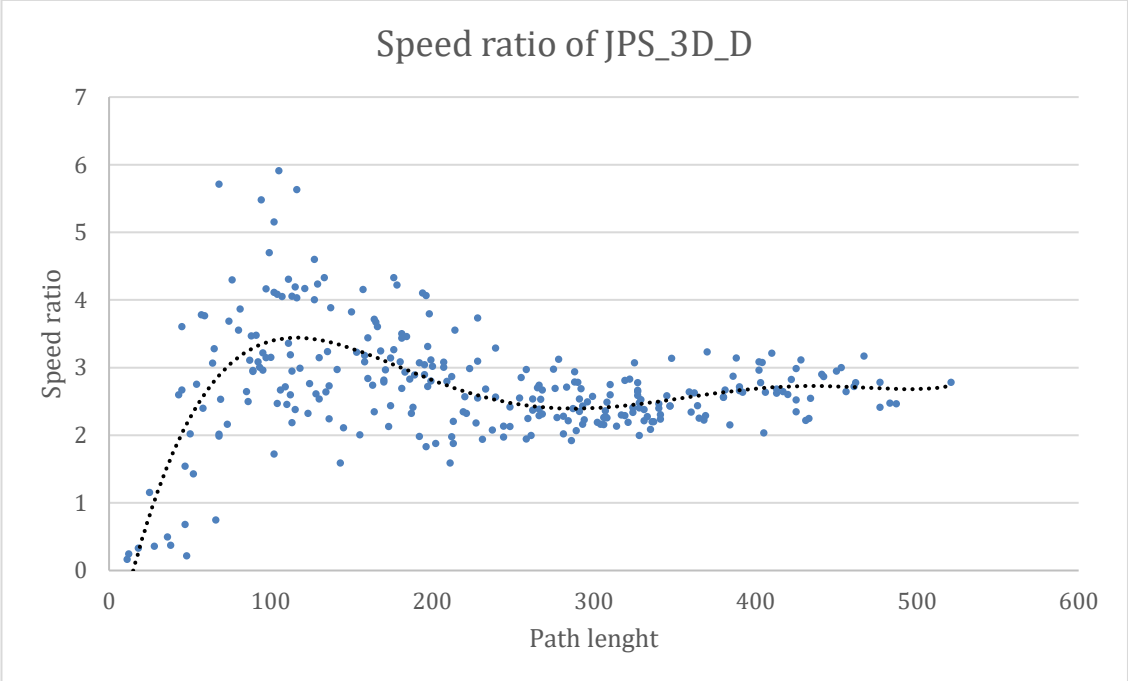

Figure 42. Speed ratio of the JPS_3D_D.

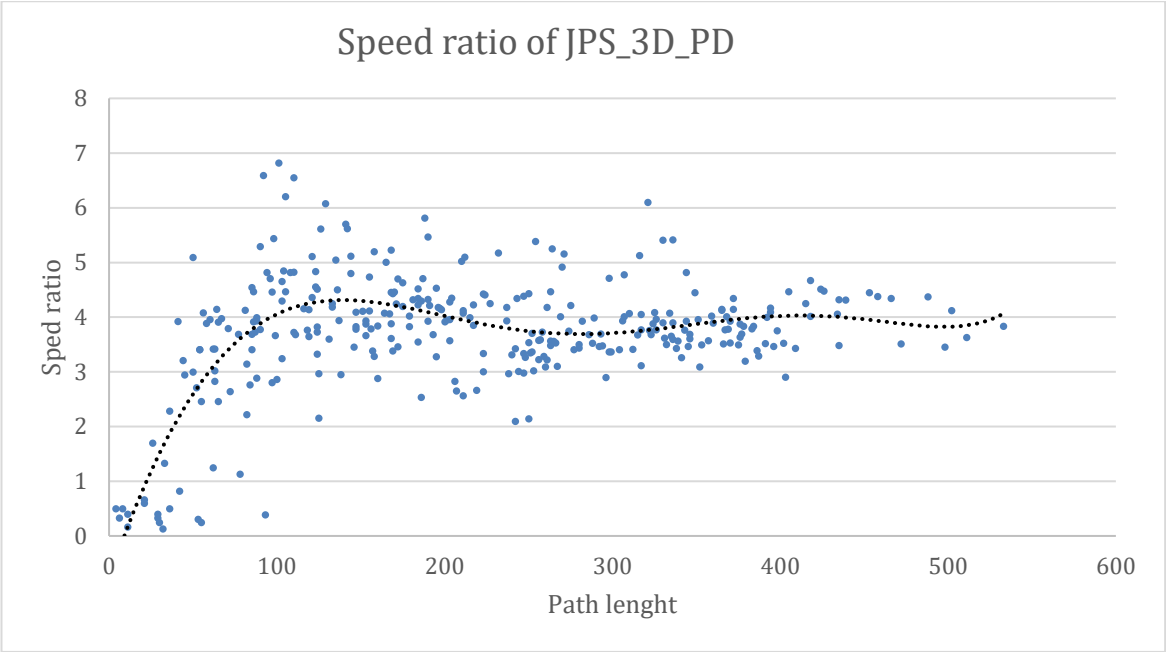

Figure 43. Speed ratio of the JPS_3D_PD.

8. CONCLUSIONS AND FUTURE WORK

The motivation of the thesis was to research the possibilities to implement a 3D pathfinding algorithm principles used in the JPS algorithm. The research reveals that these principles are viable in a 3D environment and it is possible implement the algorithm. The adapted JPS_3D algorithm reaches the same results as the A* algorithm. However, the performance of JPS_3D is non-linear and environment dependent, especially on a large empty space the solution is unfeasible. Analyzing the operation of the algorithm reveals that plain JPS_3D algorithm has some serious drawbacks. The pruning on a three-dimensional space causes excessive recurrence. The pruning optimization method can mitigate that recurrence issue slightly. Possible reason for that is that the actual implementation (the program code) is not enough optimized.

The other developed code optimization method limits the recursion depth. That approach is relatively effective and simple to implement. The method significantly improves the performance of the algorithm, additionally the performance is more linear, and it provides adequate performance on typical pathfinding environments. Using both optimization methods simultaneously is also a viable solution, even though the recursion depth optimization seems to be the more effective method.

Implemented optimization solutions were developed for the research purposes to get an idea how the efficiency of the pathfinding can be improved. Even though the results are interesting and promising, further research is needed. Generally, code optimization is a challenging task as it requires balancing between the computational and memory resources. After all, the question is about how much the optimization method requires and how much it can save those resources.

The possible future work is to research how to improve those optimization solutions. For instance, the pruning optimization is a complex task and its implementation require further research and development. Based on the tests results, the recursion depth optimization is the most promising method. The possibility to select the recursion depth limit dynamically and heuristically requires further research.

REFERENCES

[1] M. Gonzalo, Gómez-Martín and M. Antonio, Artificial Intelligence for Computer Games, Springer, 2011.

[2] P. E. Hart, N. J. Nilsson and B. Raphael, ""A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in , July 1968.," IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968..

[3] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik,* no. 1, pp. 269-271, 1959.

[4] A. Botea, M. Muller and J. Shaeffer, "Near Optimal Hierarchical Path-Finding,," Department of Computing Science, University of Alberta Edmonton, Alberta, Canada, 2004.

[5] I. Millington and J. Funge, "Artificial Intelligence for Games," 2009, p. 263.

[6] D. Harabor and A. Grastien, "Online Graph Pruning for Pathfinding on Grid Maps," in *25th National Conference on Artificial Intelligence*, San Francisco, 2011.

[7] A. Brodnik, . A. Lopez-Ortiz, V. Raman and A. Viola, Space-Efficient Data Structures, Streams, and Algorithms, Springer, 2013.

[8] A. &. K. S. Nash, "Any-angle path planning," *AI Magazine. 34,* pp. 85-107, 2013.

[9] M. H. a. R. M. Karp, "A dynamic programming approach to sequencing problems," *Journal for the Society for Industrial and Applied Mathematics,* 1962.

[10] Z. Nagy, Artificial Intelligence and Machine Learning Fundamentals:, 2018.