Proceedings of the Fifteenth International Symposium on Combinatorial Search (SoCS 2022)

The Jump Point Search Pathfinding System in 3D

Thomas K. Nobes, Daniel D. Harabor, Michael Wybrow, Stuart D.C. Walsh
Faculty of Information Technology, Monash University, Australia
{thomas.nobes, daniel.harabor, michael.wybrow, stuart.walsh}@monash.edu

Abstract

The ability to quickly compute shortest paths in 3D grids is a technological enabler for several applications such as pipe routing and computer video games. The main challenge is how to deal with the many symmetric permutations of each shortest path. We tackle this problem by adapting Jump Point Search (JPS), a well-known symmetry breaking technique developed for fast pathfinding in 2D grids. We give a rigorous reformulation of the JPS pathfinding system into 3D and we prove that our new algorithm, JPS-3D, is optimality preserving. We also develop a novel method for limiting scan depth during jump operations, which can further reduce search time. Experimental results show significant improvements versus online A* search and previous attempts at generalising JPS. We demonstrate that searching with adaptive scan limits can yield additional speedups of over an order of magnitude.

Introduction and Related Work

Three-dimensional (3D) pathfinding is a problem found in many practical applications, such as robotics, pipe-routing and video games. In each of these settings, practitioners value paths that are as short as possible and which can be computed as fast as possible. But computing optimal 3D paths is known to be NP-hard (Canny and Reif 1987). This result has motivated consideration of a variety of alternative solution strategies. Typically, these involve creating a graph to approximately represent the 3D space, and then running an optimal search algorithm, such as A*, to solve the (approximate) 3D problem as effectively as possible.

Several discretisation methods appear in the literature. In some 3D video games for example, agents have height but actually plan on a 2D mesh (Noonchester 2019). This *pseudo-3D* approach often produces reasonable solutions but is incomplete in general. The Sparse Voxel Octree (SVO) is another discrete data structure used in 3D robotics and game development (Schwarz and Seidel 2010). SVOs partition the 3D space into eight octants, which themselves are recursively partitioned if they contain any obstacle. The result is a representation that has high resolution around obstacles and low resolution in large open regions. Current SVO planners (Brewer 2019; Muratov and Zagarskikh 2019) are complete but they may produce poor quality solutions, as each path is defined in terms of octant centres. A related approach, *voxel grids*, partitions the 3D space uniformly. Paths computed with voxel grids often have much smaller detours than SVO, and are thus more desirable, especially for 3D games (Beig et al. 2019; Silva, Reis, and Grilo 2019; Alain 2018) where agents need to appear intelligent.

Yet there are several difficulties associated with pathfinding on voxel grids. First, the size of the state space is often very large; current benchmarks (drawn from real 3D games) have up to *hundreds of millions* of voxel states per map (Brewer and Sturtevant 2018). Second, the branching factor is also quite large. Each voxel has up to 26 successors, corresponding to each of the possible 3D grid moves. Third, an optimal algorithm such as A*, when applied to a voxel grid, spends substantial time expanding and exploring equivalent symmetric paths. These paths are identical to one another except for the order in which individual grid moves appear (i.e., only one is required; the rest are redundant).

In 2D grids, symmetries are handled using specialised pruning algorithms such as Jump Point Search (JPS) (Harabor and Grastien 2011). Among all possible solution paths (and partial paths), JPS prefers those where diagonal moves appear as early as possible. Other equivalent-cost paths, where diagonals appear later, are pruned. Multiple works in the literature adapt JPS to 3D but each one has various drawbacks and limitations. For example, Min, Ruy, and Park (2020) apply JPS to the problem of 3D pipe-routing but describe only a 6-connected search. Liu et al. (2017), Zhang (2021) and Zhang, Zhang, and Low (2021) consider a 26-connected grid for unmanned aerial vehicle (UAV) planning. However the computed solutions are *corner-cutting* paths, which require further post-processing before they can be passed to UAV agents for actual execution (see Figure 1). Moreover, all of these works conduct experiments on relatively small maps with only hundreds of thousands of voxels.

In this work we make the following contributions: (i) a detailed reformulation of the JPS pathfinding system to 3D voxel grid maps, which we call JPS-3D; (ii) a theoretical result that proves searching with jump points in 3D preserves feasibility (i.e., no corner cutting) and optimality; (iii) development of an adaptive method for limiting scan depth that further improves performance. We run experiments on recent voxel-world benchmarks from the literature and record up to one order of magnitude improvement over baseline A*.

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

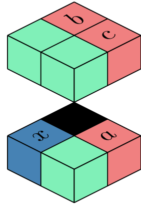

Figure 1: Example of corner-cutting on a $2\times2\times2$ grid; we show valid (green) and invalid (red) neighbours of current node x due to an obstacle (black). In the standard 2D case, nodes a and b are invalid due to sharing their common edge with an obstacle. For 3D, the move from x to c that passes through a vertex shared by an obstacle is also invalid. If we allow corner-cutting, all neighbours would be valid moves.

Notation and Terminology

We consider point to point pathfinding in an undirected uniform-cost 3D grid map. Each map comprises $x(width) \times y(height) \times z(depth)$ number of distinct voxels. Each voxel (equiv. node) is either blocked or traversable and has up to 26 adjacent neighbours. A transition from one voxel to another is called a *move*, denoted using the vector notation $\vec{m}$. A move can be in one, two or three dimensions. In particular, there are up to six one-dimensional transitions, each with a cost of 1. We call these *straight* moves. Similarly, there are up to 12 two-dimensional transitions, each with a cost of $\sqrt{2}$. We call these *2D diagonal* moves. Finally there are up to 8 three-dimensional transitions, each with a cost of $\sqrt{3}$. We call these *3D diagonal* moves. A move $\vec{m}$ is considered *feasible* if: (i) the origin and destination voxel are both traversable and; (ii) the resulting vector does not intersect the edges or corner points of any adjacent obstacle (see Figure 1 for an example); i.e., we disallow *corner-cutting* transitions.

A path $\pi = \langle n_0, n_1, ..., n_k\rangle$ is a cycle-free walk starting from a node n_0 and ending at node n_k. Equivalently, a path can be described as a sequence of moves that together allow an agent to transition from node n_0 to node n_k: i.e. $\pi = \langle n_0, \vec{m}_1, ..., \vec{m}_k\rangle$. We sometimes use the path algebra $n' = n + k \times \vec{m}$, which means that node n' is reached from node n after k applications of move $\vec{m}$. We say that the *length* of a path is equal to the number of constituent moves. The *cost* of a path is the sum of its individual move costs.

When searching for a 3D path, with A* or JPS-3D, we require an admissible estimate of the true cost-to-go from a given search node to the goal node. The three-dimensional distance between a node n and node n' on a 26-connected grid is composed of distances Δx, Δy and Δz in each respective axis. Let $d_{max} = max(\Delta x, \Delta y, \Delta z)$, $d_{min} = min(\Delta x, \Delta y, \Delta z)$, and $d_{mid} = \{\Delta x, \Delta y, \Delta z\}\backslash\{d_{max}, d_{min}\}$. The *voxel distance* between nodes n and n' is subsequently calculated as:

$$h(n, n') = (\sqrt{3} - \sqrt{2})d_{min} + (\sqrt{2} - 1)d_{mid} + d_{max}.$$

Jump Point Search in 3D

Jump Point Search (JPS) is a 2D grid pathfinding algorithm that combines the best-first expansion strategy of A* search with a recursive successor pruning strategy called *diagonal-first*. Among all optimal 2D paths JPS considers only those where diagonal moves appear as early as possible. Other paths, having the same optimal cost but diagonal moves which appear later, are pruned. In JPS-3D we generalise this idea and explore the larger 3D state space by taking 3D diagonal moves as early as possible, then 2D diagonal moves and finally straight moves. Our approach has two main components: *pruning rules*, that specify which successors to prune and which to keep, and *jumping rules*, that specify how to apply the pruning strategy recursively and when to stop. We borrow and extend terminology and definitions from Harabor and Grastien (2011), repeated here for convenience.

Pruning Rules: Given a node x reached via a parent node p, we prune from the neighbours of x any node n for which one of the following rules applies:

1. there exists a path $\pi' = \langle p, y, n\rangle$, or simply $\pi' = \langle p, n\rangle$ that is strictly cheaper than the path $\pi = \langle p, x, n\rangle$;
2. there exists a path $\pi' = \langle p, y, n\rangle$ with the same cost as $\pi = \langle p, x, n\rangle$, but π' has an earlier diagonal move than π.

Figures 2a, 2b and 2c illustrate this pruning procedure. We refer to the remaining neighbours after pruning (white) as *canonical* neighbours. Figures 2d, 2e and 2f further show that obstacles can lead to newly necessary neighbours that modify the list of successors for x, which we call *forced* successors.

Definition 1. *(Harabor and Grastien 2011) A node $n \in neighbours(x)$ is* forced *if:*

1. *n is not a canonical neighbour of x*
2. *there exists a cheaper* ***or equivalent (new in 3D)*** *cost path $\pi' = \langle p, y, n\rangle$ to the path $\pi = \langle p, x, n\rangle$ through x, that is not valid due to the step from p to y being blocked by an obstacle.*

Jumping Rules: JPS applies to each forced and canonical neighbour of the current node x a simple "jumping" procedure; the objective of which is to replace each neighbour n with an alternate successor n' that is further away. This strategy speeds up optimal search by selectively expanding only certain, interesting nodes, which we refer to as *jump points*.

Definition 2. *(Harabor and Grastien 2011) Node y is the* jump point *from node x, heading in direction $\vec{d}$, if y minimises 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 is has at least one neighbour whose evaluation is forced according to Definition 1.*
3. *The current heading direction $\vec{d}$ is a diagonal move and there exists a node $z = y + k_i\vec{d_i}$ which lies $k_i \in N$ steps in some canonical direction $\vec{d_i}$ s.t. z is a jump point from y by condition 1 or 2.*

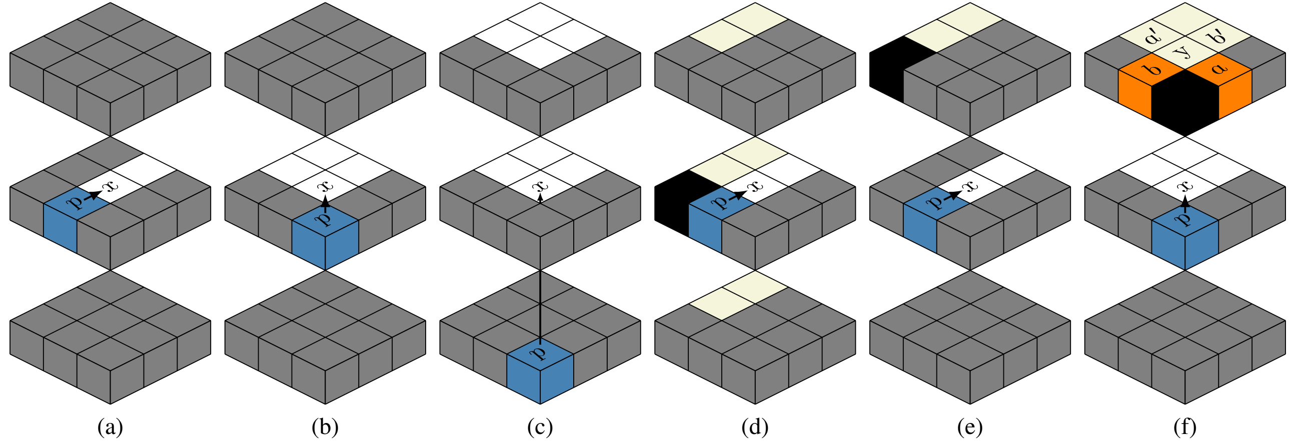

Figure 2: Each sub-figure represents a $3\times3\times3$ grid around the current node x. Canonical (white) and pruned (grey) neighbours of x reached from parent p (blue) via the corresponding move; (a) straight; (b) 2D diagonal; and (c) 3D diagonal. Forced neighbours (beige) of x either reached via a straight move due to an obstacle (black) that is (d) adjacent or (e) diagonal to p, or reached via (f) a 2D diagonal move. Either location a or b (orange) being the sole obstacle forces the corresponding neighbour a' or b' respectively by eliminating the equivalent diagonal-first path $\pi = \langle p, y, a' \rangle$ or $\pi = \langle p, y, b' \rangle$.

We now summarise how neighbour pruning is used in JPS' recursive grid-scanning procedure, adapted for 3D, and we subsequently illustrate the advantages of this search strategy.

Observe in Figure 2a that pruning for a straight move reduces the number of successors of x to a single node n. JPS exploits this property by immediately exploring n, and doing so recursively until we either (i) hit an obstacle, or reach a node n' which (ii) has a forced neighbour or (iii) is the goal itself. In case (i), all nodes along the failed path are discarded and no successors are added to the OPEN list. In case (ii) and case (iii), node n' is generated as direct successor of x. Notice how this allows the search to *jump*, from x to n', without expanding any intermediate nodes (from x to n').

Figure 2b demonstrates that 2D diagonal pruning yields three canonical neighbours. We continue to take additional 2D diagonal steps only in the case that recursion along both straight neighbours produces failed paths. This ensures that we do not miss any potential optimal turning points. A further example shown in Figure 3. Here, we continue to take 2D diagonal steps while straight recursions hit obstacles to the East and the edge of the map to the North. When recursing East from n', we identify that n'' has a forced neighbour around the obstacle and towards G. Thus, we stop scanning and return n' as a jump point.

Figure 2c demonstrates a 3D diagonal move which produces seven canonical neighbours: three straight, three 2D diagonals and one 3D diagonal. In this case, we first recurse in each canonical straight and canonical 2D direction (also inner recursions for the 2D diagonals). Only if none of these recursions yields a jump point (or the goal) do we take the next 3D diagonal step.

By jumping (see Algorithm 1), JPS is able to move quickly across the map without generating intermediate nodes in the OPEN list. This both (i) reduces the number of operations, and (ii) reduces the number of nodes in the OPEN list, effectively making each list operation itself cheaper (Harabor and Grastien 2012). Notice that JPS-3D upholds the properties of JPS; search is performed entirely online, involves no pre-processing, and has no memory overhead.

Optimality

In this section, we prove that for each optimal cost path in a 3D gridmap, there exists an equivalent cost path which can be found by only expanding jump point nodes during search. Our result closely follows the proof of the identical proposition for 2D gridmaps laid out by Harabor and Grastien (2011). We derive this result by describing the process to obtain a symmetric alternate for each optimal path. We represent this path as a series of contiguous segments, and further prove that each *turning point* along this path is a jump point.

Definition 3. *A* turning point *is any node n_k along a path where the previous move $\vec{m}_k$, from n_{k-1} to n_k, is a different direction to the next move $\vec{m}_{k+1}$, from n_k to n_{k+1}. A turning point is* optimal *if the cost of $\vec{m}_k$ and $\vec{m}_{k+1}$ together equal the cost of a cheapest path, from n_{k-1} to n_{k+1}.*

We describe a turning point at node n_k equivalently as a transition from node n_{k-1} to node n_{k+1} by the moves $\vec{m}_k$ and $\vec{m}_{k+1}$, i.e. $\langle n_{k-1}, \vec{m}_k, \vec{m}_{k+1} \rangle$. Along an optimal path we can encounter only optimal turning points. These can be constructed using the following move types: $\{\vec{m}_k, \vec{m}_{k+1} \in \{S, 2D, 3D\}\} \setminus \{\vec{m}_k = \vec{m}_{k+1} = 3D\}$ for straight (S), 2D diagonal (2D) or 3D diagonal (3D) moves. Note that a turning point requires the *direction* of travel to be different, rather than the dimension of travel (i.e. both moves can be straight, for example, as long as the directions themselves are different such as moving North after moving East). We exclude 3D-to-3D turning points as these are suboptimal and can never appear on any optimal path. Other turning points of type S-to-S, S-to-2D, S-to-3D, 2D-

to-2D and 2D-to-3D are also suboptimal, unless there exists no cheaper alternative (e.g., these can appear when an optimal path bends tightly around an obstacle).

We now define an equivalence between jump points and the turning points that appear along those paths which have a property called *3D Diagonal-First* (3DDF).

Definition 4. *A path π is* 3DDF *if it does not contain a turning point $\langle n_{k-1}, \vec{m}_k, \vec{m}_{k+1}\rangle$ that can be replaced by a turning point $\langle n_{k-1}, \vec{m}_{k+1}, \vec{m}_k\rangle$ such that the cost of π remains unchanged, matching one of the following cases:*
Case 1: $\vec{m}_k = S, \vec{m}_{k+1} = 2D$.
Case 2: $\vec{m}_k = S, \vec{m}_{k+1} = 3D$.
Case 3: $\vec{m}_k = 2D, \vec{m}_{k+1} = 3D$.

Given an arbitrary 3D path π we can repeatedly apply Definition 4 to derive a symmetric alternative, π', which has the same cost as π but where at least one 3D or 2D move appears sooner. When the procedure is no longer applicable the path π' is guaranteed to be optimal (it has the same set of moves as π and therefore the same cost) and it is guaranteed to have the 3DDF property (since no 3D or 2D move can appear sooner). The transformation algorithm is a conceptual device only. We adapt from the 2D version of Harabor and Grastien (2011) (full details contained therein).

Lemma 1. *(Harabor and Grastien 2011) Each turning point along an optimal* **3DDF path** *π' is also a jump point.*

Proof. Let n_k be an arbitrary turning point node along π'. **We consider three general cases for reasoning about n_k. We then describe briefly how each possible turning point must also be a jump point according to respective cases.**

Case 1: *Violation that π' is optimal.*
We know that there must be an obstacle adjacent (directly or diagonally) to both n_{k-1} and n_k which forces a detour. If this were not the case, we would observe that $dist(n_{k-1}, n_{k+1}) < dist(n_{k-1}, n_k) + dist(n_k, n_{k+1})$, which contradicts the fact that π' is optimal. We conclude that n_{k+1} is a forced neighbour of n_k such that by the second condition of Definition 2, n_k is a jump point.

Case 2: *Violation that π' is 3DDF.*
Turning points belonging to this case represent a turn from a lower-dimensional move m_k to a higher-dimensional move m_{k+1}. We know that there must be an obstacle adjacent (directly or diagonally) to both n_{k-1} and n_k. If this were not true, we could obtain an alternate, equivalent-cost path that does not pass through n_k by swapping the order of the moves m_k and m_{k+1}, such that the higher-dimensional move is taken first. This contradicts the fact that π' is 3DDF. Since π' is guaranteed to be 3DDF, we derive the fact that n_{k+1} is a forced neighbour of n_k. By the second condition of Definition 2, we conclude that n_k is a jump point.

Case 3: *Either the goal is reachable by a series of straight or 2D diagonal steps, or π' has additional turning points.*
If the goal is reachable by a series of one of only straight or 2D diagonal steps, we conclude that n_k has a jump node successor that satisfies the third condition of Definition 2. If n_k is instead followed by another turning point n_l, then that turning point must already correspond to either Case 1 or Case 2 accordingly, and by the argument for each case, n_l is also a jump point. We again conclude that n_k has a jump point successor that satisfies the third condition of Definition2 such that n_k is also a jump point.

S-to-S and 2D-to-2D: Both correspond to Case 1. In the absence of an obstacle, the S-to-S detour is more expensive than the direct $\sqrt{2}$ move from n_{k-1} to n_{k+1}. Similarly, the 2D-to-2D detour would be more expensive than the path that took a 3D and straight move with cost $1 + \sqrt{3}$. As π' is optimal, we conclude that n_{k+1} must be forced from n_k.

S-to-3D and 2D-to-3D: Both relate to Case 2. In the absence of an obstacle, the S-to-3D could be replaced by a 3D-to-S and the 2D-to-3D by a 3D-to-2D turning point respectively. Since π' is 3DDF, the turning points must be forced.

S-to-2D: and 2D-to-S: There are two possibilities; n_{k+1} is either (i) reachable or (ii) not reachable via a 3D diagonal move from n_{k-1}. If reachable, both detours are suboptimal compared to a direct 3D move from n_{k-1} to n_{k+1} (Case 1). If unreachable, S-to-2D could be replaced by a 2D-to-S, which violates the fact that π' is 3DDF (Case 2). For 2D-to-S, however, we observe behaviour related to Case 3, where if there is a later turning point n_l, it could be of types S-to-S, S-to-2D, or S-to3D. Alternatively, the goal is reachable by a series of straight steps. By the arguments for each, n_l is a jump point.

3D-to-S and 3D-to-2D: Both relate to Case 3. The later turning point n_l could be of types S-to-S, S-to-2D, or S-to-3D and of types 2D-to-S, 2D-to-2D, or 2D-to-3D respectively. By the arguments for each, n_l must be a jump point. □

Algorithm 1: Function $jump$

Require: x: initial node, $\vec{d}$: direction, s: start, g: goal, L scan limit, $\sum\sigma$: penalty sum
1: $\vec{d}_r \leftarrow getRecDir(\vec{d}, \vec{d}_{i\in\{1,2,3\}})$
2: $\sigma \leftarrow calcPenalty(\vec{d}, \vec{d}_r)$ ▷ σ: penalty
3: $\sum\sigma \leftarrow \sum\sigma + \sigma$
4: $n \leftarrow step(x, \vec{d})$
5: **if** n is an obstacle or is outside the grid **then**
6:     **return** *null*
7: **if** $n = g$ **then**
8:     **return** n
9: **if** $\exists\, n' \in neighbours(n)$ s.t. n' is forced **then**
10:     **return** n
11: **if** $\vec{d}$ is diagonal **then**
12:     **switch** $\vec{d}$ **do**
13:         **case** 2D
14:             $A \leftarrow \{1, 2\}$
15:         **case** 3D
16:             $A \leftarrow \{1, 2, 3, 4, 5, 6\}$
17:         **for all** $i \in A$ **do**
18:             **if** $jump(n, \vec{d}_i, s, g, \sum\sigma)$ is not *null* **then**
19:                 **return** n
20: **if** $\sum\sigma > L$ **then**
21:     **return** n
return $jump(n, \vec{d}, s, g, \sum\sigma)$

Theorem 1. *(Harabor and Grastien 2011) Searching with*

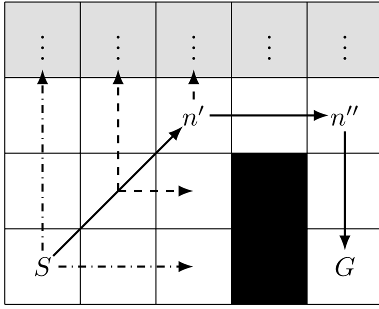

Figure 3: Example of JPS scanning. Strong lines indicate eventual jumps. Dashed lines indicate recursive straight scans before following diagonal steps. Dash-dotted lines show other directions scanned from the start S.

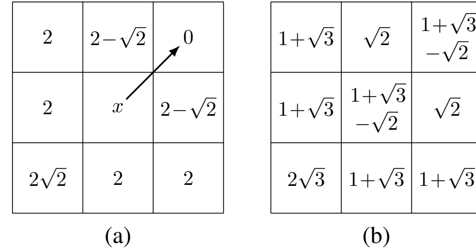

Figure 4: Penalties for each neighbour n of current node x relative to node r in the recommended direction (arrow) Northeast. The 3×3 horizontal plane (a) around x, and (b) above and below x (same values) are shown. Each penalty σ_n is calculated as $\sigma_n = h(x,n)+h(n,r)-h(x,r)$. Note that r has zero penalty.

jump point pruning always returns an optimal solution.

This proof now follows identically to the proof in Harabor and Grastien (2011). We only summarise briefly below, see the original paper for further details.

Proof. Given an arbitrary optimal path π and its symmetric 3DDF equivalent π', we will show that every turning point along π' is expanded optimally when searching with jump point pruning.

Divide π' into adjacent segments $\pi' = \pi'_0 + \pi'_1 + ... + \pi'_n$, whereby each subpath $\pi'_i = \langle n_0, n_1, ..., n_{k-1}, n_k \rangle$ is composed only of moves in the same direction (e.g., only "up" or "down", etc.). Thus, aside from the start and goal, each node at the beginning of a segment must be a turning point.

Since each segment π'_i consists only of moves in a single direction, we can jump from n_0 to n_k. Though intermediate expansions may be necessary along this subpath, we have guaranteed the path from n_0 to n_k is optimal. Since π' is also 3DDF, Lemma 1 determines that each turning point along π' must also be a jump point. As such, every turning point must be expanded during search, leaving only the start and goal nodes. The start node is trivially expanded at the start of search, and is treated specially; every direction is canonical. The goal is a jump point by definition. $\square$

Adaptive Scanning Limit

A main advantage of JPS over A* is the ability to move large distances across the grid very quickly. However, the recursive scanning procedure used by JPS to achieve this advantage has its own drawbacks, as we illustrate in Figure 3. Here, we identify a jump point at node n' due to the forced neighbour for node n''. During its diagonal recursion step, JPS will perform lengthy scanning of the grid area North. But these scans are entirely redundant, since no grid tile in this direction can possibly belong to the optimal path. In other words JPS spends substantial time pruning *surplus nodes* that could never be expanded. A* by comparison, supposing a well informed heuristic is available, will quickly expand nodes along the optimal 2D diagonal. The example motivates us to consider strategies for limiting the JPS scan depth. In particular we aim to balance two competing considerations: (1) stop scan operations sooner, to avoid exploring parts of the grid which are unlikely to contain the optimal path; and (2) reduce the number of expansions, by jumping to interesting nodes in fewer steps.

Methodology

Seeking to optimally limit the scan depth on a map-to-map basis requires prior knowledge of the domain, and likely instance-specific paths. A preliminary approach is to bound the scan by a constant step limit. This has been suggested in prior works including Harabor and Grastien (2012) and is implemented as Bounded JPS (BJPS) in Sturtevant and Rabin (2016). This method is difficult to parameterise due to the constant limit being completely uninformed about path length and the current jump direction relative to the goal.

In this work we develop an alternative method that dynamically computes a scanning limit L based on the h-value estimate of the current node n. The formulation is as follows:

$$L = (t-1) \cdot h(n),$$

where $h(n)$ is the usual distance-to-go from node n and t is a slack parameter, which controls the total amount of deviation. However, as not all jump directions are equally promising, we do not apply L uniformly. Instead, we apply a penalty σ for each step taken during a JPS grid scan. The value of σ is the difference in cost between the proposed jump direction and the direction which minimises the distance toward the target, according to the h-value function. Figure 4 shows an example.

During a scan we therefore require that the sum of penalties at each recursive step i does not surpass L; i.e., $\sum_1^i \sigma \leq L$. Notice that there is no penalty for moving in a recommended direction, and so the scan is always able to proceed unimpeded to the goal if possible (cf. stopping every k steps, as occurs with a constant limit). Exceeding the scan limit meanwhile stops the recursion immediately and produces a (pseudo-) jump node. Our strategy is optimality-preserving since each stopped scan may continue later, when the f-value of its (pseudo-) jump point becomes minimum.

The effectiveness of our proposed approach depends partly on the accuracy of the heuristic function. In this work we use

Figure 5: Example maps from Warframe benchmarks; (a) A1map and (b) C1map.

the voxel distance heuristic which has small average error on our benchmark set. We use this heuristic to compute a grid-distance estimate and a corresponding set of recommended grid moves (i.e., we compute the number of straight, 2D and 3D moves required to achieve the estimated cost). During a scan we do not penalise moves in these directions as long as there are steps remaining to be taken in that respective direction. When the scan exceeds the number of steps in a recommended direction we introduce a penalty. The penalty is always calculated relative to the closest recommended direction with remaining steps. Scan halts immediately if there are no remaining steps in any recommended direction.

To set the slack parameter, we measure (offline) the heuristic error for each instance in our benchmark set; i.e., the difference between the h-value of the start node and the optimal path cost. We run experiments with both the measured median and mean heuristic error, $t = 1.008$ and $t = 1.025$ respectively, as well as other less conservative values of $t = 1.5$ and $t = 2$. When the slack parameter $t = 1$, the scan is most tightly bound; no deviation is allowed from the recommended path. In this setup the behaviour of JPS-3D behaviour closely resembles A* search, due to having an effective step size of 1 outside of grid-best directions. However, we retain the advantage of unbounded scan towards the goal. For $t = 2$, we allow any number of scan steps up to the point that the sum of their individual penalties surpasses the estimated distance to the goal $h(n)$ at node expansion. Tuning the slack parameter t online based on instance-to-instance heuristic-error is an interesting avenue for future work.

Experimental Setup

We implement JPS-3D in C++ and we compare against a baseline implementation of 3D A*. Both algorithms are derived from WARTHOG, a freely available pathfinding library[1], and therefore share a variety of common data structures.

To evaluate performance we use a set of 44 voxel grids which range in size from 50 to 500 million voxels (Brewer and Sturtevant 2018). These maps are taken from *Warframe*, a popular online multiplayer game published by Digital Extremes in 2013. Each map is associated with 10,000 valid start-target pairs, making a total of 440,000 instances. The instances are selected at random but from among the set of voxels within 5 steps of an obstacle, so as to promote interesting, non-direct paths. We do not report results on the four variations of the C-maps (named) within the data-set due to impractical runtimes: individual instances can take up to hundreds of seconds to solve. This leaves 400,000 instances for testing. Our test machine is an Intel Xeon 8260 with 268.55 GB of RAM running Ubuntu 20.04 LTS.

[1]https://bitbucket.org/dharabor/pathfinding

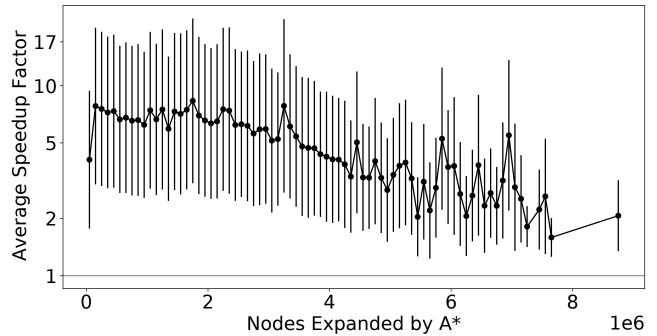

Figure 6: Average search time speedup of JPS-3D and an adaptive scanning limit with slack parameter $t = 1.5$. Bucketed by nodes expanded by A*, a measure of problem difficulty. Error bars indicate standard deviation within buckets.

Results

We discuss results in terms of the time taken to solve a problem and the number of nodes expanded with and without graph pruning. In each case we report the relative *speedup* of JPS-3D over A*. A search-time speedup value of 2 indicates the solve time was twice as fast, whereas a node-expansion speedup of 2 indicates that half the number of nodes were expanded. In both cases, a higher speedup is better. We further report the number of nodes scanned across algorithmic variants with and without limits on scan depth.

Our main result is Figure 6, which compares A* to the best version of JPS-3D, which features an adaptive scanning limit and a slack parameter $t = 1.5$. Here, we plot the average search time speedup across all Warframe instances. We group the instances into buckets, according to the number of A* node expansions, and we plot the averages of each bucket. Error bars indicate standard deviation within each bucket. We choose to bucket problem instances by the number A* nodes expansions since this gives a good measure of problem difficulty. Another often-used approach is to bucket instances according to path length. We avoid this as the variability between instances in a single bucket can produce misleading results. Further experimental data, for all our algorithmic variants, are reported in Table 1.

Our results in Figure 6 show that JPS-3D consistently outperforms A* and can be near an order of magnitude faster, on average. We also note that both methods of limiting scan report search time speedup improvements over vanilla JPS-3D. On average, our results indicate that our novel method of adaptively limiting scan outperforms a constant scan limit. However, the relative performance of both methods is quite similar; we conclude that our method is competitive with a constant scanning limit.

One surprising result, seen in Figure 6, is that relative performance of JPS-3D decreases with problem difficulty. Usu-

Algorithm	Nodes Expanded					Nodes Scanned				Search Time (ms)				
	Q1	Med	Q3	Max	Speed	Q1	Med	Q3	Max	Q1	Med	Q3	Max	Speed
A*	462	3.7k	60k	14.6M	-	-	-	-	-	0.56	2.32	33.60	1.8e+5	-
JPS-3D	**110**	**372**	**2.6k**	**1.3M**	**33.33**	5M	27M	138M	4B	1.34	5.57	24.10	2.3e+4	2.25
SL t=1.008	216	1.3k	14.0k	10.6M	5.98	**47**	**187**	523	**32k**	0.28	1.00	9.51	2.0e+4	3.94
SL t=1.025	205	1.0k	11.7k	9.8M	6.62	**47**	188	526	**32k**	0.28	0.91	8.78	2.1e+4	4.16
SL t=1.5	156	591	7.6k	8.5M	9.55	48	203	561	5M	**0.25**	0.83	**7.57**	1.6e+4	**4.71**
SL t=2	145	593	6.9k	7.8M	13.54	86	263	637	8M	0.27	1.02	8.27	**1.3e+4**	4.46
CL=50	178	672	8.6k	10.3M	6.81	125	260	**306**	153k	0.27	**0.82**	8.61	1.5e+4	3.90
CL=100	153	551	7.1k	8.0M	9.08	245	524	750	3M	0.30	0.93	8.00	1.6e+4	4.32
CL=200	133	448	5.1k	8.6M	13.31	414	972	3k	23M	0.41	1.34	8.56	1.4e+4	4.06

Table 1: Results across Warframe benchmarks. Numbers in bold represent column-best values. SL refers to an adaptive scanning limit with slack parameter t. CL refers to using a constant scan limit of some integer value. Columns Q1 and Q3 indicate values for the first and third quartile respectively. Speed refers to the average speedups across all instances.

ally, challenging instances for A* are those which exhibit "fill in": large open areas that must be explored because A*'s heuristic makes these areas appear attractive. In these settings we would expect the relative performance of JPS-3D to increase as the pruning procedure removes larger amounts of redundant work. In the Warframe benchmark, however, most maps are composed of clusters of individual asteroids along a central axis (see Figure 5a). This geometry does not explicitly feature enclosed areas of significant size that must be explored by A* for shortcuts (C-maps are a notable exception; we discuss these shortly). Here A* exhibits a type of "fill-out" behaviour, expanding nodes on and around the asteroids in order to make progress towards the goal. Although much of this work is redundant due to symmetric paths around the asteroid, A* is still able to make reasonable progress. Once the fill-out stage is completed, A* is usually able to make fast progress toward the goal as the remaining heuristic error is often small. These observations lead us to conclude that A* is a competitive algorithm for many Warframe maps.

Another observed issue, challenging for JPS-3D, involves searching across staircase-like obstacles. In this case, the scan procedure is perpetually interrupted due to a forced neighbour around the next step. When scans are short JPS exhibits A*-like behaviour, with many single step expansions required to make progress toward the target. Asteroids within the Warframe benchmark often exhibit this feature, with few walls or long axes to scan across. We omit entirely (for JPS-3D and also A*) experimental results on C-maps, a collection of 4 large maps which were computationally impractical to solve. These maps feature a uniquely large central obstacle (see Figure 5b) which must be slowly stepped over and around. Note that this central obstacle is also hollow (compounding the issues already observed).

Discussion

Several works in the literature have attempted to adapt JPS to 3D and failed to achieve search time speedups due to the large branching factor (Muratov and Zagarskikh 2019; Brewer 2019). We are aware of four successful prior attempts in the literature to translate JPS for 3D grid maps, three of which plan in the context of unmanned aerial vehicles (UAVs). In the most closely related work, Liu et al. (2017) detail planning feasible trajectories for UAVs in 26-connected gridmaps. They provide detailed explanations for neighbour pruning in 3D, but allow *corner-cutting*. In a 3D grid map, corner-cutting refers to validating diagonal moves from one voxel to another through the vertex or edge of a neighbouring obstacle (see Figure 1). Many common applications of 3D pathfinding (e.g., robotics, UAVs and video games) require that a valid path cannot corner-cut due to the agent having size at the resolution of the gridmap. Liu et al. (2017) overcome this issue by applying a series of post-processing steps to the planned path in order to obtain feasible solutions.

To compare solution quality between our works, we implement their formulation of pruning rules as a simple augmentation to our generic 3D JPS. We run experiments on synthetic maps from the literature with a total of 400,000 instances, and find that only 21% of solutions when searching with corner-cutting are valid plans. On certain maps, this can be as low as 2%. This highlights a significant limitation of prior work; solution quality of path planning with corner-cutting is poor, and often cannot find feasible solutions.

Both Zhang (2021) and Zhang, Zhang, and Low (2021) also implement JPS in 3D for route identification for unmanned drone navigation on a 26-connected voxel grid map. Insufficient detail is provided to infer their implemented pruning rules. In any case, these works are also limited in that they plan with corner-cutting and conduct post-processing steps to improve path feasibility.

Min, Ruy, and Park (2020) apply JPS to the problem of 3D pipe-routing and describe only 6-connected search. This avoids the problems faced by corner-cutting, but is a much simpler problem that cannot obtain optimal solutions for problems with diagonal movement.

Further, experiments across these four works are conducted on relatively small maps with only hundreds of thousands of voxels. In this work, we efficiently solve 26-connected experiments with hundreds of millions of voxels. No previous reformulation of JPS to 3D has been able to satisfy all of the following; (i) preserves optimality, (ii) allows full 26-connected movement, and (iii) disallows corner-cutting.

Conclusion

Here we have introduced JPS-3D; a complete reformulation of the competitive 2D JPS pathfinding system for 3D grid

maps. Our algorithm identifies and selectively expands nodes called *jump points* based on 3D path symmetry breaking. Moving between jump points involves only travelling in a fixed direction; either straight, 2D diagonal or 3D diagonal. We prove that JPS-3D upholds search optimality through its jumping procedure in which intermediate expansions along a path between two jump points never need to be expanded.

Our work is unique among previous translations of JPS to three dimensions as our algorithm can solve realistic 3D pathfinding scenarios by disallowing corner-cutting shortcuts around obstacles. We find that using jump points in 3D produces significant search time speedups compared to A*. We further develop an adaptive method for limiting scan depth to reduce over-scanning in potentially suboptimal directions by allowing an amount of slack to deviate from the heuristic-optimal path. Searching with this method returns improved speedups that can be over one order of magnitude faster.

Moreover, these results may underestimate the potential speedup in many applications. We believe that current 3D benchmarks in the literature are non-adversarial for A* in that the 3D grid-distance estimate of path cost closely resembles the true path cost. We expect that JPS-3D speedups will substantially increase on benchmarks where heuristic estimates are less informed.

Despite these improvements, 3D pathfinding remains a difficult problem. The sheer size of these maps alone means that A* solutions expand and generate up to tens of millions of nodes. On the most adversarial instances, it can take hundreds of seconds for JPS-3D to solve. One interesting direction for further work is to extend JPS-3D to generate all, but selectively insert children into the OPEN list based on their f-cost. This idea, known as partial expansion, has been investigated in a variety of domains where problems have prohibitively large branching factors (Goldenberg et al. 2014; Felner et al. 2012). Due to the high branching factor of 3D problems, this could yield substantial performance improvements.

Acknowledgements

This research was funded by Woodside Petroleum Ltd. through the Monash University Human-in-the-Loop Analytics Graduate Research Industry Partnership. We thank our Woodside collaborators for many useful discussions and ongoing support. Daniel Harabor is partially supported by the Australian Research Council under grants DP190100013 and DP200100025, and by a gift from Amazon.

References

Alain, B. 2018. Hierarchical Dynamic Pathfinding for Large Voxel Worlds. https://www.gdcvault.com/play/1025151/Hierarchical-Dynamic-Pathfinding-for-Large. Accessed: 2022.

Beig, M.; Kapralos, B.; Collins, K.; and Mirza-Babaei, P. 2019. G-SpAR: GPU-Based Voxel Graph Pathfinding for Spatial Audio Rendering in Games and VR. In *2019 IEEE Conference on Games (CoG)*, 1–8.

Brewer, D. 2019. *Game AI Pro 360*, chapter 3D Flight Navigation Using Sparse Voxel Octrees, 273–282. CRC Press. ISBN 9780429055096.

Brewer, D.; and Sturtevant, N. R. 2018. Benchmarks for Pathfinding in 3D Voxel Space. *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 9(1): 143–147.

Canny, J.; and Reif, J. 1987. New lower bound techniques for robot motion planning problems. In *28th Annual Symposium on Foundations of Computer Science (SFCS)*, 49–60.

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Beja, T.; Sturtevant, N.; Schaeffer, J.; and Holte, R. 2012. Partial-expansion A* with selective node generation. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research*, 50: 141–187.

Harabor, D.; and Grastien, A. 2011. Online Graph Pruning for Pathfinding on Grid Maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, 1114–1119. AAAI Press.

Harabor, D.; and Grastien, A. 2012. The JPS pathfinding system. In *International Symposium on Combinatorial Search*, volume 3.

Liu, S.; Watterson, M.; Mohta, K.; Sun, K.; Bhattacharya, S.; Taylor, C. J.; and Kumar, V. 2017. Planning Dynamically Feasible Trajectories for Quadrotors Using Safe Flight Corridors in 3-D Complex Environments. *IEEE Robotics and Automation Letters*, 2(3): 1688–1695.

Min, J.-G.; Ruy, W.-S.; and Park, C. S. 2020. Faster Pipe Auto-Routing Using Improved Jump Point Search. *International Journal of Naval Architecture and Ocean Engineering*, 12: 596–604.

Muratov, T.; and Zagarskikh, A. 2019. Octree-Based Hierarchical 3D Pathfinding Optimization of Three-Dimensional Pathfinding. In *Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control*, 1–6. Amsterdam Netherlands: ACM.

Noonchester, A. 2019. 'Marvel's Spider-Man' AI Postmortem. https://gdcvault.com/play/1025828/-Marvel-s-Spider-Man. Accessed: 2022.

Schwarz, M.; and Seidel, H.-P. 2010. Fast Parallel Surface and Solid Voxelization on GPUs. *ACM Transactions on Graphics*, 29(6).

Silva, G.; Reis, G.; and Grilo, C. 2019. Voxel Based Pathfinding with Jumping for Games. In *EPIA Conference on Artificial Intelligence*, 61–72. Springer.

Sturtevant, N. R.; and Rabin, S. 2016. Canonical Orderings on Grids. In *IJCAI*, 683–689.

Zhang, N.; Zhang, M.; and Low, K. H. 2021. 3D Path Planning and Real-Time Collision Resolution of Multirotor Drone Operations in Complex Urban Low-Altitude Airspace. *Transportation Research Part C: Emerging Technologies*, 129: 103–123.

Zhang, S. 2021. *Trajectory Planning Based on Optimized Jump Point Search Results Using Artificial Potential Field in 3-D Environments*. Master's thesis, University of California, Santa Cruz.