Sub-Microsecond Grid Path Planning, at What Cost?

Mark Carlson, Daniel D. Harabor, Peter J. Stuckey

Department of Data Science and Artificial Intelligence, Monash University
mark.carlson@monash.edu, daniel.harabor@monash.edu, peter.stuckey@monash.edu

Abstract

Tree Cache is a lightweight pre-processing approach to grid path finding which works by generating a shortest path tree: from a root cell to all cells in the map. During online search Tree Cache simply follows the tree: from start and target towards the root, stopping at the first common cell. Although Tree Cache is fast, the resulting paths have no solution quality guarantees. In this paper we improve Tree Cache, in terms of speed and solution quality, by combining symmetry breaking ideas from Jump Point Search. Our new algorithm, Jump Spanning Tree Search (JSTS), can usually generate paths with low average sub-optimality in under one microsecond – up to two orders of magnitude faster than Tree Cache. We then extend JSTS to derive a new and very fast bounded sub-optimal search, which guarantees solution quality in single-digit microseconds on average. Our results establish a remarkable new level of performance in the area. In particular, we show JSTS approaches and often improves upon the *output complexity* of an idealised oracle, which simply reads off and returns a corresponding but optimal solution path.

Introduction

Shortest path problems are one of the most well-studied problem classes in Computer Science. Finding a shortest path on a grid is a ubiquitous task for many video games and is a widely-used simplification in robot motion planning, particularly in the context of multi-agent path finding. For these applications, it is important that path finding algorithms are fast and produce high-quality solutions.

Jump Point Search (JPS) (Harabor and Grastien 2011, 2014) is an optimal grid path finding algorithm which is typically two orders of magnitude faster than A* on the grid, despite not using any pre-processing. To go faster than this, pre-processing is required. Differential heuristics (Sturtevant et al. 2009) in conjunction with JPS are able to find paths several factors faster than JPS with an inexpensive pre-processing step. To go three orders of magnitude faster than A* requires expensive pre-computation techniques such as Jump Point Graphs (Harabor et al. 2019) and JPS+BB+ (Hu et al. 2021). These algorithms are the fastest optimal grid path finding algorithms in the literature at this time.

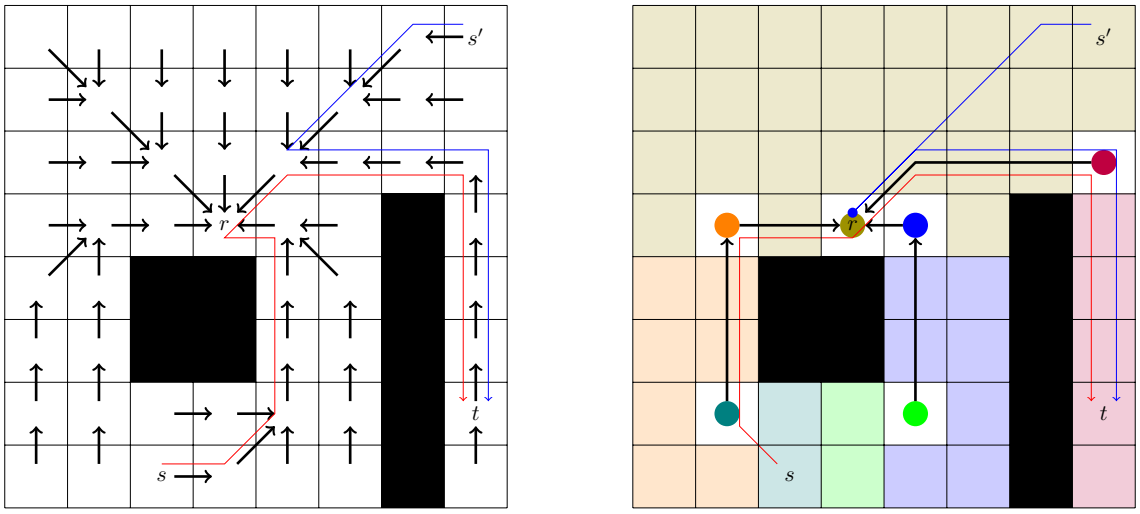
Algorithm	Time (μs)	Guarantees	Pre-proc
A*	89979	Optimal	None
JPS	536	Optimal	None
JPS+	349	Optimal	Seconds
JPS+ with DH	85	Optimal	Minutes
CH-JPD	54	Optimal	Hours
JPS+BB+	24	Optimal	Days
Tree Cache	4	Complete	Seconds
JSTS (new!)	**0.3**	Complete	Seconds

Table 1: Summary of path planning methods listed on the Grid-based Path Planning Competition (GPPC) website (GPPC Staff 2025). Time is average time per path. Pre-processing is approximate sum over all maps.

Sub-optimal algorithms trade solution quality for speed. Tree Cache (Anderson 2012) achieves four orders of magnitude speedup versus A*, but at the cost of unboundedly sub-optimal paths. Unfortunately, this is the only sub-optimal path finding algorithm in this performance class in the literature. Sub-optimal CPDs (Zhao et al. 2020) provide bounded sub-optimality at reduced pre-processing cost compared to JPS+BB+, but at a similar path finding performance level.

The Grid-based Path Planning Competition (GPPC) (GPPC Staff 2025) is a forum for evaluating state-of-the-art solvers in the area. Evaluation is undertaken on 256,000 problem instances drawn from a variety of real-world and synthetic maps. Undominated performance in the GPPC is regarded as a strong indicator of leading performance. A summary of results from the GPPC is given in Table 1.

In this paper, we present two new sub-optimal algorithms which offer significantly improved performance compared to existing methods. First, Jump Spanning Tree Search (JSTS) improves on the performance of Tree Cache by an order of magnitude (five orders of magnitude faster than A*), at only a minor cost to average solution quality. This performance level matches that of an idealized oracle, which simply reads off an optimal grid path. The second algorithm, Bounded Jump Spanning Tree Search (BJSTS), provides bounded sub-optimality guarantees at the same performance level as Tree Cache. Our results establish new benchmark standards of performance for both unbounded sub-optimal and bounded sub-optimal grid-based pathfinding.



(a) (b)

Figure 1: (a) A spanning tree rooted at r, and path returned from s to t (red) and s' to t blue. (b) A jump spanning tree for the same map: jump points and root are circled. Grid cells apart from jump points are colored by the color of their parent node. The paths returned from s to t (red) and s' to t (blue) are shown, notice how the path from s' to t is longer in the jump spanning tree.

Problem Definition

We investigate path planning on 8-connected grid maps without corner-cutting. A grid map consists of $W \times H$ cells and each cell is either traversable or non-traversable. A move $\vec{m}$ represents a transition between orthogonally-adjacent or diagonally-adjacent cells a and $b = a + \vec{m}$. A move is valid if both a and b are traversable, and we enforce no corner-cutting; diagonal moves have the additional requirement that the two cells orthogonally adjacent to both a and b are also traversable. The cost of a move $\vec{m}$ is simply the length of the move vector $|\vec{m}|$.

A path $\pi = [n_1, \ldots, n_l]$ of l steps is a sequence of grid cells where each segment $[n_i, n_{i+1}]$ is valid. A segment is valid if, for some $k \in \mathbb{Z}^+$, $n_{i+1} - n_i = k\vec{m}$ and the move $\vec{m}$ is valid at each cell $n_i + j\vec{m}, j \in 0 \ldots k-1$. In other words, a segment is valid if n_{i+1} can be reached from n_i through repeated application of a move $\vec{m}$. The cost (or length) $|\pi|$ of a path π is the sum of the costs of each segment, which is given by $k|\vec{m}|$. This representation of paths is used by the Grid Path Planning Competition (GPPC) (GPPC Staff 2025). Paths can be reversed using the operator $\textsc{Rev}(\pi)$ and concatenated using the operator $\pi_1 ++ \pi_2$. The tail operation, $\textsc{Tail}$ returns all but the first element of a path, while $\textsc{Init}$ returns all but its last element.

Tree Cache

Tree Cache (Anderson 2012) is a very straightforward approach to quickly finding paths in a undirected graph $G = (V, E)$. A root node $r \in V$ is chosen, and a shortest path spanning tree is computed in G from root r, defining for each node $v \in V$ both the parent of the node in the spanning tree $parent(v) \in V$, where $parent(r) = r$, and the depth from the root node $depth(v) \in \mathbb{N}$ where $depth(r) = 0$ and $depth(v) = 1 + depth(parent(v)), v \neq r$.

We can find a path from any node s to another node t by simply following their parent pointers to the first shared ancestor. This is formalized in Algorithm 1. A path query from s to t invokes $\textsc{TreeCache}(s, t, depth(s), depth(t), parent)$. We assume that the depth of t is greater than or equal to that of s, and if not swap the roles, recording this in rev. We step from t to its ancestor at the same depth as s, building the path p_t. The function $\textsc{Add}(p, v)$ is simply defined as $p ++ [v]$ (we will revisit this later). Once the depth of the revised target reaches that of s, we build path p_s from s and extend the path from t together step by step until they reach the same node. We then construct the full path from s to t by taking the path p_s and adding the reverse of the path p_t, reversing the result if we swapped s with t initially.

For example, in Figure 1(a) to find a path from $s = (3, 1)$ to $t = (8, 2)$, we look up their depths $depth(s) = 6$, $depth(t) = 8$. We then traverse from t to the cell two squares above at depth 6 $[(8, 2), (8, 3), (8, 4)]$. We simultaneously traverse from s and this cell, meeting at the root, and combine these paths (reversing that from t) to generate the red path $[(3, 1), (4, 1), (5, 2), (5, 3), (5, 4), (5, 5), (4, 5), (5, 6), (6, 6), (7, 6), (8, 6), (8, 5), (8, 4), (8, 3), (8, 2)]$.

This algorithm has a time complexity of $O(M)$ where M is the length of the path produced by the algorithm. The length of the path is bounded by two times the height of the constructed spanning tree, which is often significantly smaller than the number of nodes in the graph. The construction of a shortest-path spanning tree can be done quickly (in $O(|V|)$ for grid map domains using a relaxed node expansion order (Otte 2015)). However, the algorithm does not provide any bounds on the sub-optimality of the path.

Algorithm 1: Tree cache search: given start s and target t with depths d_s and d_t and tree defined by $parent$ return a path from s to t.

procedure TREECACHE($s, t, d_s, d_t, parent$)
  $rev, p_s, p_t \leftarrow false, [s], [t]$
  if $d_s > d_t$ **then**
    $rev, s, t, d_s, d_t, p_s, p_t \leftarrow true, t, s, d_t, d_s, p_t, p_s$
  while $d_t > d_s$ **do**
    $p_t \leftarrow$ ADD($p_t, parent(t)$)
    $t, d_t \leftarrow parent(t), d_t - 1$
  while $s \neq t$ **do**
    $p_t \leftarrow$ ADD($p_t, parent(t)$)
    $t \leftarrow parent(t)$
    $p_s \leftarrow$ ADD($p_s, parent(s)$)
    $s \leftarrow parent(s)$
  $p \leftarrow$ INIT(p_s) $+\!\!+$ REV(p_t)
  if rev **then return** REV(p)
  else return p

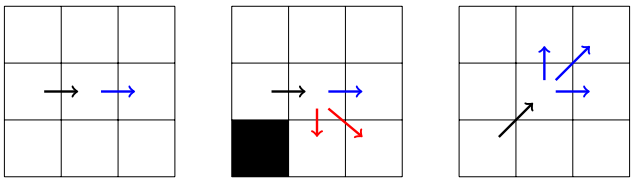


Figure 2: Unpruned moves in JPS. Red arrows move to forced successors while blue move to natural successors.

Jump Point Search

In empty grid maps many optimal paths exist between two locations, differing only in the order of diagonal and orthogonal moves. These paths are said to be *symmetric*. Jump Point Search (JPS) (Harabor and Grastien 2011, 2014) eliminates symmetries by only considering a unique *canonical path*, where all diagonal moves occur before any orthogonal move (diagonal-first). All other optimal paths are pruned. In grid maps that contain non-traversable cells the situation is more complicated. Here JPS identifies locations where an orthogonal move is allowed to follow a diagonal, because the diagonal-first path is blocked by a non-traversable cell. These locations, where the canonical ordering of moves is reset, are known as *jump points*. Thus, any optimal path on the grid can be transformed into a sequence of diagonal-first segments connecting jump points; i.e., a canonical path.

Jump points are identified by the application of neighbour pruning rules during node online search. These rules are shown in Figure 2. In empty space, orthogonal moves (left) are always followed by the same orthogonal move to the sole *natural successor*, as the other moves are either sub-optimal or not diagonal-first (and thus pruned). However, in the presence of obstacles (center), an orthogonal turn or diagonal move may be required; these moves lead to *forced successors*. Diagonal moves (right) have three natural successors: the diagonal continuation and two orthogonal moves. Because we disallow corner-cutting, diagonal moves cannot have forced successors.

Jump Point Search recursively expands nodes where all successors are natural successors. The remaining nodes, which have forced successors, are jump points and are placed on the open list. This recursive process is called *jumping* and significantly reduces the number of nodes which are placed on the open list. Further performance gains can be achieved by pre-computing the result of all possible "jump" operations. The pre-computation step is a *clearance table* which tells the distance to a next jump point in each of the eight grid directions. This algorithm is known as JPS+.

Jump Spanning Tree Search

We seek to improve performance by reducing the number of segments of paths found by Tree Cache using jump points. At a high-level, we modify the construction of the shortest-path spanning tree using the pruning rules of Jump Point Search so that $parent(v)$ refers to the parent jump point instead of the immediate parent grid cell. This reduces the height of the spanning tree, allowing TREECACHE to find paths in significantly fewer steps.

Construction

We begin by constructing a clearance table we will use to determine when diagonal-last paths are feasible. The clearance table stores the number of times each of the eight grid moves can be repeated before becoming infeasible due to a non-traversable cell. These values are easily calculated in linear time using a dynamic programming approach; see (Harabor et al. 2019) for details. We also re-use this table during search to directly connect the start to the target with either a diagonal-first or diagonal-last path, which mitigates some of the largest proportional sub-optimality.

Next, we perform a single-source shortest path search from the chosen root node using the JPS pruning rules without jumping. We make two modifications to the basic procedure to track jump point parents and avoid staircases.

Jump Point parents. When examining a cost-improving edge from p to n during the expansion of p, we determine if n is a natural successor according to the JPS pruning rules. If n is a natural successor of p and p is not the root node, then we assign $parent(n) = parent(p)$ and $depth(n) = depth(p)$, bypassing p. The effect of this is to make $parent(n)$ the nearest ancestor which had a forced successor; the parent jump point of n.

Staircase bypassing. Our jump point spanning tree often suffers from a staircase effect illustrated in Figure 3. The diagonal-first path from A to B requires many turning points and jump points, increasing the length of the path and therefore the height of the tree. To mitigate this issue we observe that the equal-cost diagonal-last path is often feasible, which in this example has a turning point at C. We implement the idea as follows. When expanding a node n with parent $p = parent(n)$, we use the clearance table to test if the direct diagonal-last path from the grandparent $parent(p)$ to n is feasible. If it is, then we assign $parent(n) = parent(p)$ and $depth(n) = depth(p)$, bypassing p. We do this in a way which does not interfere with the determination of jump point parents described above.

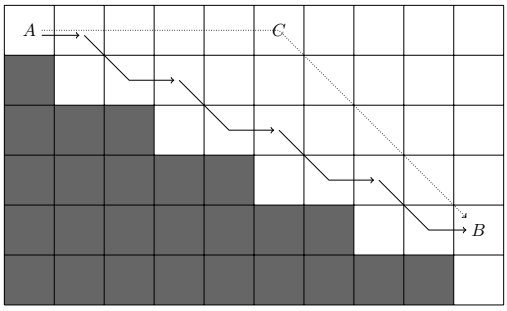


Figure 3: Staircases require many jump points to reach the bottom, and can be circumvented by using the diagonal-last dotted path instead.

Optimised Tree Representation

The trees produced by our method contain vastly more leaf nodes than internal nodes, motivating a representation that treats internal nodes specially. Our representation uses two data structures, a small array for the internal nodes of the tree storing parent and turning point and a large two-dimensional array for connecting each grid cell to an internal node storing parent, depth and turning point. The internal nodes array is often small enough to fit inside the L1 data cache.

The connection array associates three values with each grid cell v: the index of the connection node in the internal nodes array $parent(v)$, the depth of the connection node to the root $depth(v)$, and the location of the diagonal-first or diagonal-last turning point if a turn is required $turningpoint(v)$. The connection node for grid cells v corresponding to leaf nodes stores all information. For grid cells corresponding to an internal node v, the connection node is itself $parent(v) = v$. This type of connection must be checked for during search to avoid adding a zero-length segment to the final path.

The internal nodes array associates three values with each node: its location, the index of its parent node in the internal nodes array $jparent$, and the location of the diagonal-first or diagonal-last turning point $turningpoint$ if a turn is required, but no depth information. The order of the nodes in this array have a minor impact on search performance. We found that a depth-first pre-order with children ordered descending by the size of their subtree is a good choice.

In Figure 1(b) the internal (jump point and root) nodes are shown as colored circles, and the $jparent$ edges are illustrated in black. The $parent$ information for grid cells (other than internal nodes) is indicated by the background color of the cell. The shortcircuiting of jump point parents is illustrated by e.g. cell (5,1) whose parent in JPS+ would be cell (5,2), but since it is a natural successor its parent is (5,5).

Root Selection

The original Tree Cache paper suggests choosing a central node, but does not go into detail. We consider three strategies for choosing the root node: *Random*, which functions as a baseline; *Central*, which chooses the node closest to the center-of-mass of the traversable area; and *Midpath*, an advanced selection algorithm which we will describe here.

Algorithm 2: Jump Spanning Tree Search: given start s and target t and jump point tree defined by $jparent$ and parent and depth information for leafs given by $parent$ and $depth$ return a path from s to t.

procedure JSTS($s, t, jparent, parent, depth$)
  $j_s, d_s \leftarrow parent(s), depth(s)$
  $j_t, d_t \leftarrow parent(t), depth(t)$
  $p \leftarrow$ TREECACHE$(j_s, j_t, d_s, d_t, jparent)$
  if $s \neq j_s$ **then**
    $p \leftarrow$ INIT(ADD$([s], j_s)$ ++ p
  if $t \neq j_t$ **then**
    $p \leftarrow p$ ++ REV(INIT(ADD$([t], j_t)$)
  return p

Our Midpath root node selection algorithm attempts to select a topologically central node as the root by taking the midpoint of a long shortest path in the map. First, we pick a random node x and perform a single-source shortest path to find the furthest node y from x. Then we perform another single-source shortest path from y to find the furthest node z from y. We then pick the root node r to be the midpoint along the shortest path from y to z.

This strategy for picking a root node has additional benefits when used with Bounded Jump Spanning Tree Search, which is described later. During the two single-source shortest path queries, we can record in a table the distance each node is from the root of the search. This table can then be used as a Differential Heuristic (Sturtevant et al. 2009). In fact, this algorithm follows the advanced pivot selection procedure except that we do not discard the distance data for the initial random node.

Search Algorithm

Jump Spanning Tree Search (JSTS) is shown in Algorithm 2. We first find the parents j_s and j_t in the jump spanning tree for the start s and target t, we then apply TREECACHE to build the a path from j_s to j_t, then add in the links from the original start and target if they are not nodes in the jump spanning tree.

The attentive reader will have noticed that the path segment from a node v to its parent in the jump spanning tree, $[v, parent(v)]$, is not valid if it combines orthogonal and diagonal moves. To address this, we redefine ADD to add the turning point if one exists between v and $parent(v)$. So ADD$(p, parent(l))$ where l is the last node in p is defined to construct the path p ++ $[turningpoint(l), parent(l)]$ if there is a turning point from l to $parent(l)$, and p ++ $[parent(l)]$ otherwise. This ensures that the every edge between two consecutive nodes in the path is valid.

Re-examining the example in Figure 1(b) to find a path from $s = (3, 1)$ to $t = (8, 2)$ we find the depths and parents $depth(s) = 3$ and $depth(t) = 2$, $parent(s)$ is the teal jump point at $(2, 2)$, and similarly $parent(t)$ is the purple jump point at $(8, 6)$. We call TREECACHE on the jump tree returning the path connecting them, and the connect s and t to this finally returning the red path as $[(3, 1), (2, 2), (2, 5), (4, 5), (5, 6), (8, 6), (8, 2)]$.

Bounded Sub-optimal Search with Jump Spanning Trees

A key weakness of JSTS is that it can return unboundedly sub-optimal paths. Bounded Jump Spanning Tree Search (BJSTS) addresses this by using JSTS as an upper bounding heuristic in a JPS+ style (Harabor and Grastien 2014) search to provide bounded sub-optimality. BJSTS (Algorithm 3) returns a path from s to t no more than w times the length of the optimal path. It is simply a JPS+ search using a heuristic h, but it outputs paths from s to t of decreasing length, found using JSTS. We maintain an incumbent I, the node for which the path from s to I found by the search concatenated with $\text{JSTS}(I, t)$ is the lowest cost solutions found so far. Whenever we expand a node n we check whether completing the path from n to t with JSTS leads to a shorter path than the incumbent. If it does, then we set $I = n$ and output the path. Once the f bound times w reaches the incumbent path cost u, we have proven the incumbent solution is within the sub-optimality bound and finish.

This procedure is similar to CPD search (Bono et al. 2019). The main difference is that CPD search provides an accurate lower bound h (under the assumption that the CPD is computed on a graph with lower edge costs), while here we simply use another h function (in practice, the differential heuristic provided by the root node selection strategy described earlier). Because we don't get a lower bound from JSTS, we update the incumbent when expanding a node rather than when relaxing a node. Finally, we use jump point exploration since we are path planning on an unweighted grid. Like CPD search, this algorithm also provides anytime behaviour, outputting a series of feasible solutions with decreasing cost until it finds one within the bound.

The pseudo-code hides some optimisations. Since we only query $|\text{JSTS}(\cdot, t)|$ except when outputting a path, we can mark each ancestor node of t with the distance to t. Computing $|\text{JSTS}(n, t)|$ then requires only ascending the jump spanning tree to the first marked node m and adding the distance from n to m to the stored distance from m to t. The cost of these ascents can be further reduced by employing the same caching strategy used in CPD search.

Algorithm 3: Bounded Sub-optimal JSTS: return a path from s to t no more than w times the optimal path length.

procedure $\text{BJSTS}(s, t, w)$
  $open \leftarrow \{s\}$
  for $n \in N$ **do** $g[n] \leftarrow \infty$
  output $\text{JSTS}(s, t)$
  $g[s], f[s], p[s], u, I \leftarrow 0, h(s, t), [], |\text{JSTS}(s, t)|, s$
  while $open \neq \emptyset$ **do**
    $n \leftarrow \arg\min\{f[v] \mid v \in open\}$
    if $n = t$ **then return** $p[n]$
    if $g[n] + |\text{JSTS}(n, t)| < u$ **then**
      $u, I \leftarrow g[n] + |\text{JSTS}(n, t)|, n$
      output $p[n] +\!+ \text{JSTS}(n, t)$
    if $w \times f[n] \geq u$ **then return** $p[I] +\!+ \text{JSTS}(I, t)$
    $open \leftarrow open \setminus \{n\}$
    for m jump point successor of n **do**
      $q \leftarrow \text{ADD}(p[n], m)$
      if $|q| < g[m]$ **then**
        $p[m] \leftarrow q$
        $g[m] \leftarrow |q|$
        $f[m] \leftarrow g[m] + h(m, t)$
        $open \leftarrow open \cup \{m\}$
  return $\perp$

Experimental Setup

We implemented the algorithms in Rust using the mkpath pathfinding library. Full code is available at `github.com/MinusKelvin/2025-jump-sts`. Experiments were run on an Ubuntu 22.04 system running kernel 6.8.0-52 with an AMD Ryzen 9 5950X at 4.5GHz and 16GB 3200MHz DDR4 memory.

We evaluate the algorithms on the standard Moving AI grid map benchmarks (Sturtevant 2012) and Iron Harvest grid map benchmarks (Harabor, Hechenberger, and Jahn 2022). We group these into four categories: *Game*, all of the maps from games in MovingAI; *Street*, all of the city street maps in MovingAI; *Maze*, all of the synthetic maze maps in MovingAI; *Random*, all of the synthetic random maps in MovingAI; *Room*, all of the synthetic room maps in MovingAI; and *Iron Harvest*, all of the maps from the Iron Harvest grid map benchmarks.

We evaluate Jump Spanning Tree Search on three primary metrics: tree construction time, search time, and sub-optimality factor. We primarily compare it to our implementation of Tree Cache, which uses the same optimised representation and clearance table direct connection. We also compare it to an idealised oracle which simply reads off an optimal path which does not use jumps; this represents a lower bound on the performance of any algorithm which does not use some form of contraction. Search time is reported as the average time per path, which is calculated as the real time required to search all problem instances on a map divided by the number of problems and repeated 100 times. We do not attempt to measure search times for individual problem instances because the timer on our test system has resolution and overhead similar to quantity being measured, which would introduce significant error.

We evaluate Bounded Jump Spanning Tree Search primarily on the search time in relation to the requested sub-optimality bound. We compare it against JPS+ (which BJSTS decays to as w approaches 1), JPS+ using a weighted heuristic and no re-expansions,[1] and JPS+BB+. All of these algorithms use the differential heuristic obtained from the Midpath root node selection strategy. We measure the average time per path for BJSTS in the same way as for JSTS (although with only 10 repeats) as the same timer resolution issue can occur when the JSTS path is immediately returned.

[1]This may not be a correct algorithm; see (Carlson, Harabor, and Stuckey 2024). However, they did not observe any failures on unweighted grids without corner-cutting and we have not either, and their mitigation introduces some overhead.

	JSTS			JSTS w/o staircase bypass			Tree Cache			Copy Oracle	
	Search	Seg.	Subopt	Search	Seg.	Subopt	Search	Seg.	Subopt	Copy	Seg.
Game	**39**	20.9	1.206	55	40.7	1.195	537	410.1	**1.154**	47	372.5
Iron Harvest	**68**	29.0	1.133	236	248.4	1.131	3558	2971.7	**1.124**	582	2713.0
Street	**37**	13.2	1.227	65	43.4	1.220	642	511.5	**1.190**	56	471.2
Maze	594	490.1	1.027	**573**	760.0	1.026	2668	2128.7	**1.016**	531	2122.1
Room	114	66.5	1.372	**95**	81.7	1.370	583	447.7	**1.347**	43	368.7
Random	335	210.1	1.396	**251**	287.7	1.395	655	512.0	**1.381**	54	435.0

Table 2: Comparison of variants of tree based search versus an oracle copy — search or copy time (ns), average number of segments in path, geometric mean suboptimality

	Midpath		Midpath, no direct		Central		Random	
	Geomean	99th	Geomean	99th	Geomean	99th	Geomean	99th
Game	**1.206**	4.758	1.452	20.207	1.232	**4.588**	1.325	5.365
Iron Harvest	**1.133**	**2.788**	1.148	3.240	1.584	4.535	1.365	3.729
Street	**1.227**	5.121	1.614	29.565	1.230	**4.919**	1.420	6.736
Maze	**1.027**	**1.621**	1.036	1.994	1.035	1.635	1.035	1.627
Room	**1.372**	7.721	1.431	10.657	1.386	**7.622**	1.585	9.827
Random	**1.396**	**10.845**	1.412	11.986	1.400	10.886	1.631	14.348

Table 3: Sub-optimality factors (geometric mean and 99th percentile) using different tree root heuristics

Experimental Results

Jump Spanning Tree Search

We measured the average time per path, average number of path segments, and geometric mean sub-optimality of JSTS, JSTS without staircase bypassing, Tree Cache, and an idealized oracle which simply reads off an optimal path without jumps. The results are shown in Table 2. As expected, JSTS significantly reduces both the search time and number of path segments compared to Tree Cache at a small increase in average path cost, up to 5%. On all maps except for narrow-corridor mazes, the average time per path is well under a microsecond. However, some of these results are also fairly surprising.

For the Game, Street, and Iron Harvest maps, JSTS is faster than the idealised oracle, even nearing an order of magnitude on Iron Harvest, and is competitive against it on Maze. This is a result of jumping, which significantly reduces the number of segments in the returned path by an order of magnitude (two in the case of Iron Harvest). On Random maps, jumping is largely ineffective.

For the synthetic Maze, Room, and Random maps, staircase bypassing results in an increase in running time despite a reduction in the average number of path segments. In fact, in all benchmarks the JSTS variant without staircase bypassing has lower time per path segment. This is especially surprising as both methods use the same runtime search implementation and data structures. Since our strategy for ordering the internal nodes array in the optimised tree representation places nodes in a staircase contiguously in memory, we conjecture that this is a result of favorable CPU caching behaviour.

Tree Cache is much slower than JSTS since the number of segments in its paths are much higher. Notably Tree Cache leads to slightly lower sub-optimality, since it can find a shared ancestor further from the root, since its not restricted to jump points. This is illustrated in Figure 1 for the path s' to t which is longer using JSTS. Similarly removing staircase bypassing can improve sub-optimality because the shared ancestor can be further from the root. But the increase in sub-optimality for JSTS is not large.

Table 3 shows the sub-optimality factor of JSTS, both as geometric mean and at the 99th percentile, comparing various choices for choosing the root of jump tree. We compare 4 variations: picking a Random root, picking a root as Central as possible (to the mean cell position), and Midpath choosing a root by performing two shortest path calculations as described in Root Selection above, and in addition the Midpath approach where we do not check for direct paths between start and target. The results show that the mean sub-optimality is not high for any graph, never exceeding 1.4 with Midpath. The choice of root does have an appreciable effect, with the Midpath approach clearly improving on the others, although not by much except on the larger Iron Harvest maps. Checking for a direct path usually has a larger impact on mean sub-optimality than root node choice, but its real value is shown by the 99th percentile sub-optimality which are drastically improved. The Random maps have the worst sub-optimality, since they do not have open space which defeats the direct path check, although the optimal path is often close to a direct path. The Mazes have the best sub-optimality, as they are tree-like and the sub-optimality comes from the corridor width.

Table 4 gives the average and maximum time to construct the jump spanning tree data structures for the map classes we consider, using the Midpath and Central root node selection strategies, as well as the average number of traversable cells for each map class. Construction time is more or less linearly related to number of traversable cells, the large Iron Harvest

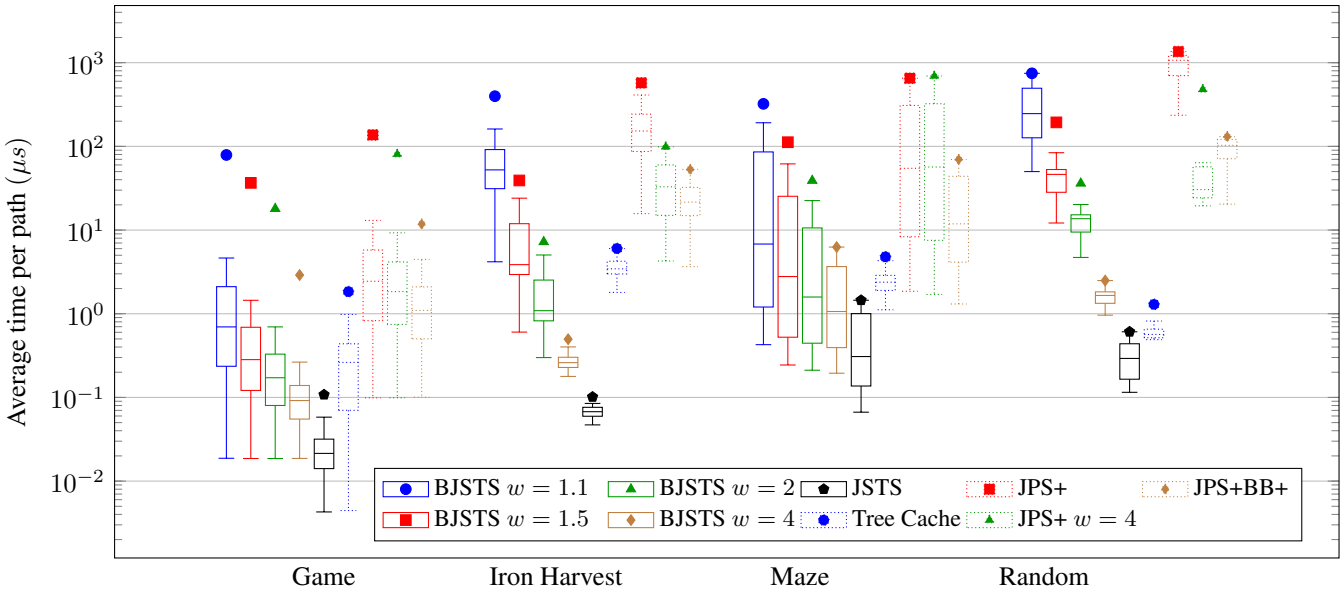


Figure 4: Box plots of average path construction time (with max value shown as a dot) for BJSTS for different values of w over the 4 different map types compared with JSTS, JPS+, JPS+BB+, and a variant of JPS+ using weighted A* expansion ordering.

	Midpath		Central		
	Mean	Max	Mean	Max	Cells
Game	0.027	0.312	0.020	0.220	64.9k
Iron Harvest	4.464	14.494	3.012	9.733	6.8M
Street	0.164	0.455	0.118	0.321	347k
Maze	0.081	0.093	0.057	0.065	208k
Room	0.091	0.108	0.064	0.066	233k
Random	0.093	0.116	0.071	0.088	186k

Table 4: Time to construct jump spanning trees (seconds) and the average number of traversable cells on each map.

	$w = 1.1$	$w = 1.5$	$w = 2$	$w = 4$
Game	49.4%	75.3%	87.2%	97.6%
Iron Harvest	35.7%	82.8%	94.2%	99.4%
Street	49.3%	79.7%	88.9%	97.5%
Maze	47.3%	74.6%	85.8%	96.2%
Room	19.2%	62.8%	79.2%	94.3%
Random	30.2%	64.3%	77.5%	92.4%

Table 5: Percentage of BJSTS instances which immediately return the JSTS path.

maps pay for more cache misses. The additional overhead from the more advanced root node choice strategy is about 40%. For most benchmark maps the trees are constructed in well under a second, while the much larger Iron Harvest maps take several seconds. These construction times are fast enough to perform when loading the map, but without an efficient repair strategy online updates to the map remain a challenge, unless they are infrequent.

Bounded Jump Spanning Tree Search

Bounded Jump Spanning Tree Search (BJSTS) ameliorates the major disadvantage of JSTS, that there is no bound on the sub-optimality. This of course comes at a bound-dependent performance cost; we give results for $w \in \{1.1, 1.5, 2, 4\}$. Figure 4 shows box plots of average time per path across maps in each of four different map classes: Game, Iron Harvest, Maze and Random. We restrict to four to make the plot fit, and these classes cover the best and worst cases for BJSTS. We compare against JPS+, JPS+ using a weighted A* expansion order with $w = 4$, and JPS+BB+ as described in the experimental setup. We also include results for JSTS and Tree Cache for context.

As the w bound for BJSTS increases the average time for it to find a solution within the bound decreases significantly, even for Mazes where JPS+ $w = 4$ fails to outperform optimal JPS+. This is particularly interesting as even with only a 10% relaxation BJSTS is competitive with the best optimal approach JPS+BB+ which requires significant pre-processing. When using the same $w = 4$ bound, BJSTS outperforms JPS+ with a weighted A* expansion order by more than an order of magnitude in all cases. Notably, with $w = 1.5$, BJSTS often matches the performance of Tree Cache except on the Random maps.

Table 5 shows the proportion of instances for BJSTS which immediately return the path JSTS(s, t). This explains the low average time per path of BJSTS, since at $w = 4$ almost all instances simply call JSTS. Even for the strongest bound $w = 1.1$ a significant proportion of instances immediately return. This is possible because we use a strong heuristic, obtained for free during root node selection, which provides large lower bounds. Table 6 shows the effect of using this heuristic in comparison to the traditional octile distance heuristic in terms of nodes expanded by BJSTS. The

	$w = 1.1$	$w = 1.5$	$w = 2$	$w = 4$
Game	70.1%	76.6%	78.1%	77.4%
Iron Harvest	66.6%	82.8%	86.6%	90.3%
Street	56.4%	60.4%	53.1%	36.7%
Maze	88.3%	94.0%	96.5%	98.7%
Room	37.0%	35.6%	32.3%	27.0%
Random	61.2%	78.3%	84.1%	90.0%

Table 6: Percentage reduction in nodes expanded due to differential heuristic versus octile heuristic.

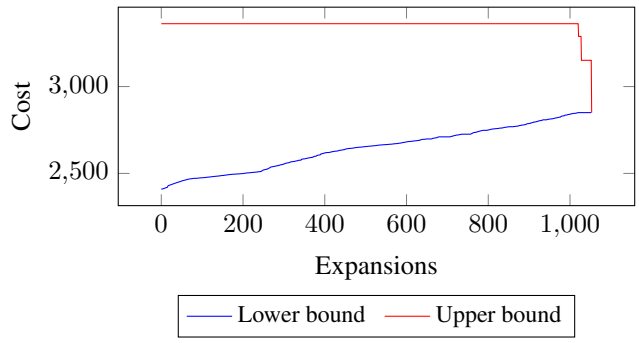


Figure 5: Lower and upper bounds vs expansions for BJSTS on an Iron Harvest problem instance.

differential heuristic often reduces node expansions by several factors. Interestingly the effect is not monotonic in w as when the octile distance heuristic is sufficient to immediately return there can be no node reduction.

Figure 5 shows the anytime behaviour of BJSTS as a plot of the upper and lower bounds as nodes are expanded on one Iron Harvest problem instance which is run to optimality. Interestingly, BJSTS does not find many different candidate paths on its way to finding one within the bound w; in this case, only two improvements on the initial incumbent are found before optimality is achieved. Instead, most of the effort is in raising the lower bound, which reduces the gap from 1.40 to 1.18 before the first improved path is found, which further reduces the gap to 1.15. Because the spanning tree is constructed as a shortest path tree from the root, improvements must result from JSTS finding a closer shared ancestor, which cannot happen more times than the height of the tree. This underscores the importance of using an accurate heuristic with BJSTS.

Conclusion

Jump Spanning Tree Search is a blindingly fast approach to grid-based path planning – up to five orders of magnitude compared to conventional grid-based A* search. The *cost* for this level of performance is solutions that are 20% worse than optimal on average (and 3-5x worse than optimal at the 99th percentile, on grids from real application domains). Alternatively JSTS can be employed as part of a bounded suboptimal search that provides near-optimal cost guarantees at up to four orders of magnitude performance improvement (cf. grid-based A*).

JSTS requires only linear time pre-processing, typically just a few seconds even for large maps. That suggests JSTS could be an appealing algorithm for a variety of applications – most notably computer games including those where the map changes only infrequently (one could simply compute the jump tree on-demand and on-the-fly). It is worth noting that in modern games the majority of pathfinding calls are actually used by the "AI" for action planning in the medium and long-term (cf. for immediate movement and navigation). That means most paths are never followed; see e.g., Goal Oriented Action Planning (Orkin 2006), where the agent must evaluate distances to points of interest in order to formulate a high-level plan. By speeding up these calls JSTS gives back more time for the AI to make better high level decisions. JSTS sets a new performance standard in this area.

A question that arises is whether we can do better than the approach here, since each JSTS path finding query is completed on the order of a few hundred clock cycles; i.e., near the output complexity of an idealised oracle that *simply reads off an optimal path*. Future work could investigate how to improve path quality without losing too much speed. One possible direction involves the use of multiple trees for improving path quality, an idea which has been previously suggested for Tree Cache (Bader 2016).

Acknowledgments

This research is supported by an Australian Government Research Training Program (RTP) Scholarship and Australian Research Council Grant DP200100025.

References

Anderson, K. 2012. Tree Cache. In *Proceedings of the International Symposium on Combinatorial Search*, volume 3, 203–203.

Bader, S. 2016. *Pathfinding with Trees*. Master's thesis, University of Basel.

Bono, M.; Gerevini, A. E.; Harabor, D. D.; Stuckey, P. J.; et al. 2019. Path planning with CPD heuristics. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, 1199–1205.

Carlson, M.; Harabor, D.; and Stuckey, P. J. 2024. Avoiding Node Re-Expansions Can Break Symmetry Breaking. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, 20–27.

GPPC Staff. 2025. Grid-based Path Planning Competition. https://gppc.search-conference.org/.

Harabor, D.; and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI conference on artificial intelligence*, volume 25, 1114–1119.

Harabor, D.; and Grastien, A. 2014. Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, 128–135.

Harabor, D.; Hechenberger, R.; and Jahn, T. 2022. Benchmarks for pathfinding search: Iron harvest. In *Proceedings of the international symposium on combinatorial search*, volume 15, 218–222.

Harabor, D. D.; Uras, T.; Stuckey, P. J.; and Koenig, S. 2019. Regarding jump point search and subgoal graphs. In *International Joint Conference on Artificial Intelligence 2019*, 1241–1248. Association for the Advancement of Artificial Intelligence (AAAI).

Hu, Y.; Harabor, D.; Qin, L.; and Yin, Q. 2021. Regarding goal bounding and jump point search. *Journal of Artificial Intelligence Research*, 70: 631–681.

Orkin, J. 2006. Three states and a plan: the AI of FEAR. In *Game developers conference*, volume 2006, 4. Citeseer.

Otte, M. W. 2015. Modifying Dijkstra's Algorithm to Solve Many Instances of SSSP in Linear Time. *Univeristy of Colorado Boulder Aerospace Engineering Sciences Faculty Contributions*.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2): 144–148.

Sturtevant, N. R.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-Based Heuristics for Explicit State Spaces. In *IJCAI*, 609–614.

Zhao, S.; Chiari, M.; Botea, A.; Gerevini, A. E.; Harabor, D.; Saetti, A.; and Stuckey, P. J. 2020. Bounded suboptimal path planning with compressed path databases. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 333–341.