Optimal Pathfinding on Weighted Grid Maps

Mark Carlson[1], Sajjad K. Moghadam[2]
Daniel D. Harabor[1], Peter J. Stuckey[1], Morteza Ebrahimi[2]
[1]Department of Data Science and Artificial Intelligence, Monash University
[2]University of Tehran
mark.carlson@minuskelvin.net
{daniel.harabor, peter.stuckey}@monash.edu
{sajjad.moghadam, mo.ebrahimi}@ut.ac.ir

Abstract

In many computer games up to hundreds of agents navigate in real-time across a dynamically changing weighted grid map. Pathfinding in these situations is challenging because the grids are large, traversal costs are not uniform, and because each shortest path has many symmetric permutations, all of which must be considered by an optimal online search. In this work we introduce Weighted Jump Point Search (JPSW), a new type of pathfinding algorithm which breaks weighted grid symmetries by introducing a tiebreaking policy that allows us to apply effective pruning rules in symmetric regions. We show that these pruning rules preserve at least one optimal path to every grid cell and that their application can yield large performance improvements for optimal pathfinding. We give a complete theoretical description of the new algorithm, including pseudo-code. We also conduct a wide-ranging experimental evaluation, including data from real games. Results indicate JPSW is up to orders of magnitude faster than the nearest baseline, online search using A*.

Introduction

Practical approaches to 2D path finding in games or robotic applications often represent the navigable area as grid map. Path finding then becomes finding a sequence of grid cells to visit to move from a start cell to a target cell.

A main challenge in grid pathfinding is symmetry, since there are usually many paths of identical cost from a start to target. Jump Point Search (JPS) (Harabor and Grastien 2012) is an online and optimal algorithm that prunes symmetric grid paths and which can be orders of magnitude faster than standard A* search (Hart, Nilsson, and Raphael 1968). Although widely used in computer games and robot navigation, JPS is limited to uniform cost maps with only traversable or non-traversable tiles.

In this paper we extend JPS to grid maps with terrains of different costs. We show that a straightforward extension of JPS to this new domain incurs substantial overhead costs. We then introduce a new and efficient algorithm, Weighted Jump Point Search (JPSW), to effectively prune the search space. We show that JPSW is optimal and online and up to an order of magnitude faster than its nearest competitor, a conventional and online A* search.

Problem Model

We have a grid map of $H \times W$ cells. For each cell we have an associated terrain cost $t \in \mathbb{R}^+$. A *move* is a transition from one grid cell to another adjacent grid cell. Each move is represented as a vector $\vec{m}$ and has a corresponding direction and a magnitude. We distinguish between orthogonal moves $\vec{o}$ of length 1 and diagonal moves $\vec{d}$ of length $\sqrt{2}$.

Cost model: In our work the cost of moving from one grid cell to another adjacent grid cell is equal to the magnitude of the corresponding move vector multiplied by the weighted average terrain cost, considering all the tiles intersected during the move action. For example, the cost of an orthogonal move, such as from s to o in Figure 1 (left), is equal to the average terrain cost of cells s and o. The cost of a diagonal move meanwhile, such as s to d in Figure 1 (left), is the weighted average of all four cells touching the diagonal move, s, u, o and d, multiplied by $\sqrt{2}$ (since any non-point agent will intersect all four cells). We use notation $|\langle s, d\rangle|$ to denote the (weighted) cost of move from s to d. Other cost models are also possible and have been considered elsewhere; e.g., multiply the magnitude of the move vector by the terrain type of the destination tile, or taking the average of the source and destination tiles only. The ideas presented here are applicable to these other cost models. A special case, which we refer to as *uniform cost*, are grids where every cell is either traversable with unit terrain cost or non-traversable with infinite cost.

A path p of m moves is given by a sequence of m grid cells $\langle n_0, \ldots, n_m\rangle$ where n_i is adjacent to n_{i+1} by move $\vec{m}_i$. The path length $|p| = \sum_{i=1}^{m-1} |\langle n_i, n_{i+1}\rangle|$, the sum of the costs of each move along the path. We also define paths by a starting node and sequence of moves, e.g. $n_0 + k\vec{m}$ is a path of k moves in direction $\vec{m}$ starting from cell n_0.

Related Work

Pathfinding is an intensely studied topic in the research literature, with many efficient algorithms having been suggested. In this section we give a brief overview of related works in this area. A main observation is that, despite a wealth of research, none of the currently available techniques are able to accelerate optimal weighted grid pathfinding in online settings, where terrain costs are subject to change. In computer game applications, where such setups are common and opti-

mal paths are demanded, the best option remains A* search with a simple heuristic.

WRP: Our work is related to the Weighted Region Problem (WRP), a well known and well studied topic in the area of Computational Geometry. In WRP we are given planar subdivision of a 2D map, with costs assigned to each of the distinct regions. The objective is to find a path from start point s to target point t such that the sum of its weighted Euclidean segments is minimum, among all st paths. WRP has eluded an exact solution method for decades and recent results suggest one may never be found (De Carufel et al. 2014). Bounded suboptimal approximations exist but they run in high-order polynomial time (Mitchell and Papadimitriou 1991). Unbounded suboptimal methods for WRP also exist, but are often complex to implement or not practical for performance sensitive applications such as games; see (Tran, Dinneen, and Linz 2020) for a recent example.

Optimal Grid Search: Rasterised grids provide a simple way to approximate a weighted planar subdivision. Among their many advantages, grids are easy to implement, fast to update and fast to search. Grid-optimal solutions are also often close to Euclidean-optimal solutions, with higher resolution grids giving closer approximations. A variety of conventional speedup techniques based on graph augmentation are applicable here; e.g., Contraction Hierarchies (Geisberger et al. 2008). Though very fast these algorithms rely on precomputed auxiliary data, which is invalidated when the grid changes. Unfortunately dynamic changes can be frequent in computer video games, and repair or reconstruction of auxiliary data can be too expensive to perform online.

Another type of speedup technique involves precomputed heuristic functions. Examples include single-dimension estimators, such as Differential Heuristics (Sturtevant et al. 2009) and FastMap (Cohen et al. 2017), as well as APSP oracles, such as Compressed Path Databases (Bono et al. 2019) and Hub Labels (Delling et al. 2014). These approaches all assume that traversal costs can never fall below some floor (e.g., free-flow cost in a road network). Provided the assumption holds, dynamic changes cannot invalidate lower-bounding data. Unfortunately this assumption often does not hold in computer video games, where environments are destructible (in which case the heuristic becomes equivalent to popular but not very accurate relaxations; e.g., octile- or Euclidean-distance).

Approximate Grid Search: A variety of works which employ abstraction can handle weighted grids. Perhaps the most relevant to our work is DTA (Sturtevant et al. 2019), which can find a near-optimal path on a dynamic map with multiple terrains, including for agents with different cost models. Being suboptimal, the paths may require further post-processing before they can be used for navigation.

Jump-Point Search

Jump-Point Search (JPS) (Harabor and Grastien 2012) is an optimal and online pathfinding algorithm for uniform-cost grids. There are two key ingredients: pruning rules, which eliminate redundant and symmetric successors, and jumping rules, which skip over irrelevant cells where the search cannot branch. We give a brief description of this algorithm.

Pruning rules: when considering a move $\vec{m}$ from a cell x to a potential successor cell n, JPS takes into consideration p, the parent cell of x. The first rule prunes $\vec{m}$ if the path $\langle p, x, n\rangle$ is longer than another path $\langle p, y, n\rangle$ or sometimes simply $\langle p, n\rangle$. The second rule considers symmetrical paths, and prunes $\vec{m}$ if there is another path $\langle p, y, n\rangle$ in which a diagonal move occurs earlier. The set of successors that are not pruned when there are no non-traversable cells nearby are called *natural successors*. Any successor that is not a natural successor is called a *forced successor*.

Jumping rules: when a cell is expanded, instead of adding its direct successors to OPEN, JPS recurs aiming to find alternative successors in the same direction, that are farther away. When considering an orthogonal direction $\vec{o}$, the jumping procedure scans in the direction of the move until one of the three stopping conditions is met:

1. The target cell is reached.
2. A cell with at least one forced successor is reached.
3. The move is no longer possible due to an obstacle.

In cases 1 and 2 the cell that was reached is added to OPEN. In case 3 no successor is generated.

Jumping in a diagonal direction $\vec{d}$ is similar but requires more work. Before each diagonal step JPS requires two recursions, in direction $\vec{o_1}$ and $\vec{o_2}$ s.t. = $\vec{o_1} + \vec{o_2} = \vec{d}$. If one of these jumps succeeds, then the cell reached by the orthogonal jump is added to OPEN and the diagonal scan continues. Notice that JPS is *diagonal-first*: i.e., along the path to each generated node diagonal moves appear as early as possible. By comparison, A* search considers every possible (and symmetric) move ordering to each generated node. Pruning away such *non-canonical successors* reduces the branching factor per node. Jumping meanwhile avoids OPEN list operations for nodes with a branching factor of zero or one. For these reasons, compared to A* search, JPS can be an order of magnitude faster (and more) (Harabor and Grastien 2014).

Weighted Jump-Point Search

JPS eliminates many redundant grid paths through *symmetry breaking*. Symmetric paths consist of the same set of moves in different orders. The diagonal-first strategy distinguishes between these paths and eliminates all but one. In the weighted grid case we achieve a similar effect using a tie-breaking policy called *orthogonal last*.

Consider the problem of finding a path π with the minimum *tiebroken cost*, a lexicographically ordered $(g, |\vec{m}|)$ tuple where g is the cost-thus-far and $\vec{m}$ is the last move action along path π. We will prune a move to a node only if we show that there exists another path to that node which has strictly lower tiebroken cost. However, a minimum tiebroken cost path could have a prefix which is not minimum tiebroken cost, which if pruned would prune the path transitively. Theorem 1 shows that every reachable node has a minimum tiebroken cost path to it which cannot be pruned by our pruning condition, and so our algorithm is optimal.

Theorem 1 *For all reachable nodes n, there exists a path π to n with minimum tiebroken cost such that all prefixes of π have minimum tiebroken cost.*

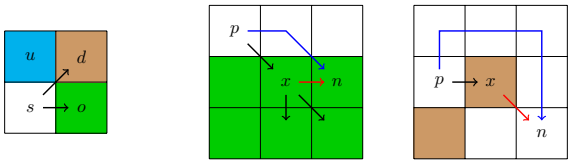

Figure 1: White has cost 1, green 2, brown 10, and cyan 0.1. (Left): The diagonal move $\langle s, d \rangle$ has cost $13.1/4\sqrt{2}$, and the orthogonal move $\langle s, o \rangle$ has cost 1.5. (Middle) and (Right): The blue path proves that the red move is unnecessary.

Proof: We will proceed by induction on the path length, g.

Base case. The empty path to the start node has no prefixes and has minimum tiebroken cost $(0, 0)$.

Inductive case. Suppose the statement holds true for all paths with $g < g_n$. Let n be a reachable node distinct from the start node with $g = g_n$. Because n is reachable and not the start node, there exists a minimum tiebroken cost path to n ending with a move $\langle p, n \rangle$ from another reachable node p with $g < g_n$, but which may have prefixes that are not minimum tiebroken cost. However, by the inductive hypothesis, there exists a path π to p which has minimum tiebroken cost and for which every prefix is minimum tiebroken cost. Extending π with the move $\langle p, n \rangle$ produces a path to n with the minimum tiebroken cost $(g_n, |n - p|)$, and every prefix of this path has minimum tiebroken cost. □

We will begin by generalising the pruning and jumping rules of JPS to weighted grid maps under this framework.

Pruning Rules We generalise the pruning rules that apply when considering a move $\vec{m}$ from a cell x with parent p to a potential successor cell n by analysing the 3×3 neighbourhood of x. The generalised pruning rule prunes the move $\vec{m}$ if any path inside the neighbourhood from p to n has a lower tiebroken cost than the path $\langle p, x, n \rangle$. The effects of this rule can be seen in Figure 1. We call the set of unpruned moves the neighbourhood successors of x. This rule encapsulates both rules from JPS due to the definition of tiebroken cost, however it is phrased in terms of the type of the final move. On uniform-cost girds, the new rule turns out to be equivalent to JPS' diagonal-first rule. In weighted grids the new rule is better as the paths involved may be complex, as illustrated in the rightmost example in Figure 1.

To calculate the neighbourhood successors for a cell given its 3×3 neighbourhood and parent cell, we use an (online) Dijkstra's algorithm to find minimum tiebroken cost paths to each cell in the neighbourhood. The cells whose shortest path ends with a move from the centre cell then correspond to moves in the neighbourhood successor set. Note that with this strategy the search requires an additional tiebreaking rule to prefer paths coming from the centre cell. This prevents the rule from excluding an orthogonal-last optimal path.

While each Dijkstra search is very quick, it could still be undesirable to run the computation anew, each time we need to identify the successors of a node. If the set of cell costs that could appear on the weighted grid map is known in advance, we could precompute the neighbourhood successors for all possible 3×3 neighbourhoods and parent directions. However, such a database would require $\mathcal{O}(9^k)$ space and time for k different terrain types, so this quickly becomes impractical, even when taking advantage of rotational and mirror symmetry. Additionally, since weighted grid maps are generally not random, most of the entries in the database would never be used.

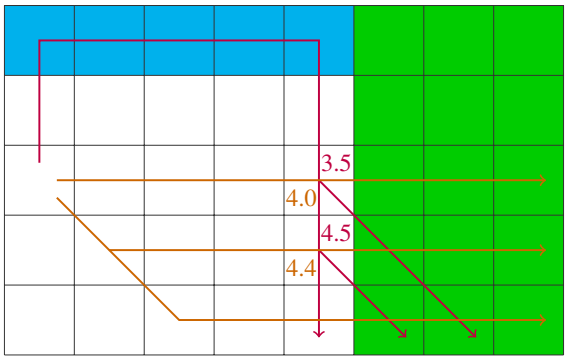

Figure 2: Allowing the orange scans to continue through the terrain transition prevents the purple path from discovering that it is dominated partway down the transition. This generates redundant scans into the high-cost green region.

Instead, we use a cache (that takes advantage of rotational symmetry) and only compute the neighbourhood successors for the neighbourhoods that are actually observed during the search. This has the additional advantage of being able to handle sets of terrain costs that are not known in advance.

Jumping Rules Like in JPS, when a cell is expanded, we run a jumping procedure to find alternative successors that are farther away. However, we use more aggressive stopping conditions. Instead of stopping at the first cell with at least one forced successor, we stop at the first cell whose 3×3 neighbourhood contains more than one terrain type. This condition results in us always enqueueing nodes that occur on the border of terrains.

Note that the *natural generalisation* of the JPS jumping rule might be to continue until there is a successor not in the same direction as the current jump. However, while this may prevent the generation of some nodes, this can result in extra unnecessary and avoidable work. While an orthogonal jump crossing a light-to-heavy terrain transition would continue through the transition, there may be an alternative path which runs parallel to the terrain transition. Without the nodes from the orthogonal jumps, the alternative path may not discover it is dominated and generate nodes at the terrain transition anyways. Figure 2 illustrates this case.

Overscanning

A problem that may show up in JPS is *overscan*, in which two diagonal jumps produce overlapping orthogonal scans. This can happen when an obstacle may be navigated around in more than one direction. While this results in quadratic worst-case time complexity, the patterns that produce this overscan are uncommon and do not affect very many scans when they occur.

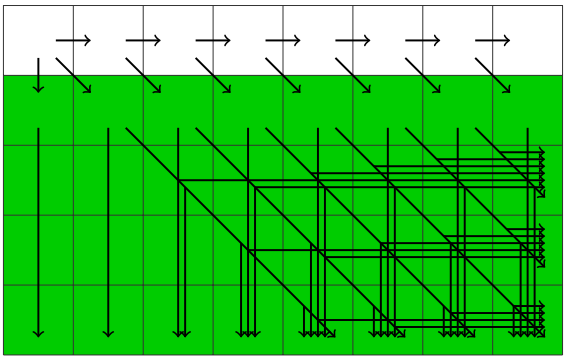

Figure 3: Scanning pattern entering the heavy ($w = 2$) green region from the top-left.

Unfortunately, the opposite is true for JPSW. Even a simple terrain transition, as shown in Example 1, is enough to produce a catastrophic amount of overscan.

Example 1 Consider the map shown in Figure 3. Using the pruning and jumping rules the search enqueues each node in the first and second row. When we dequeue the second cell of the second row, we begin a diagonal scanning exploration that explores the entire uniform terrain area down and to the right. We add any jump points discovered into the queue. When we later dequeue the third cell of the second row, we begin a diagonal scanning exploration down and to the right which overlaps with the previous scan in all but the leftmost column. For the rightwards scans, we will need to adjust the g-value of all jump points rediscovered, since the path is shorter. For the downwards scans, we find that the path is longer for all of the jump points rediscovered, making the scans useless. The same applies for the fourth cell in the second row, etc. Overall we repeatedly scan the same area, and repeatedly discover the same jump points, each time with either a smaller g-value which will be further reduced during the next scan, or a larger g-value which does not affect the search. □

We present two approaches to solving this problem. The first approach uses two complimentary sets of rules, which we call Diagonal Branch Pruning and Prospective g Pruning, to eliminate irrelevant scans during a diagonal jump. The second approach uses a jump cache to reduce the cost of overscanning from quadratic time complexity to amortised linear time. Furthermore, these approaches can be combined to achieve even better performance.

Diagonal Branch Pruning In the example illustrated in Figure 3 the optimal route to every cell in the heavy area is either a completely diagonal path from the top row, or a diagonal path followed by some downwards steps for cells underneath the diagonal from the top left cell.

To avoid the unnecessary horizontal overscans we adopt the following *diagonal branch pruning rule*: If we are expanding a node n in a diagonal direction $\vec{d} = \vec{h} + \vec{v}$ and the node n does not have a successor in orthogonal direction $\vec{h}$ then during the diagonal scan we do not scan in direction $\vec{h}$.

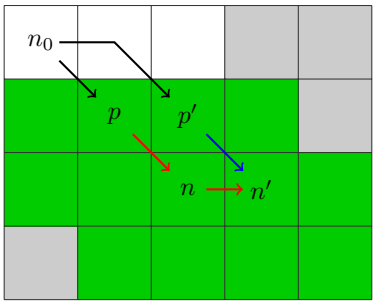

Figure 4: Illustration of the proof of Theorem 2. Since p has no orthogonal successor to p' the blue path via p' must be shorter than the red path via n. Grey terrain is irrelevant.

Theorem 2 *The diagonal branch pruning rule never removes an orthogonal last optimal path to any cell.*

Proof: Let n be a cell in uniform terrain reached from its parent p by a diagonal move $\vec{d} = \vec{h} + \vec{v}$. Consider the case where $p' = p + \vec{h}$, $n' = n + \vec{h}$, and the move $\langle p, p' \rangle$ has been pruned. See Figure 4.

Because the move $\langle p, p' \rangle$ has been pruned, we know from the pruning condition that there exists an alternative path to p' with tiebroken cost $(g_{p'}, |\vec{m}|) < (g_p + |\langle p, p' \rangle|, 1)$. No move action has a length less than 1, so this implies $g_{p'} < g_p + |\langle p, p' \rangle|$. Consider the paths $A = \langle ..., p', n' \rangle$, which extends the alternative path to p', and $B = \langle ..., p, n, n' \rangle$. The cost of A is $g_{p'} + |\langle p', n' \rangle|$ and the cost of B is $g_p + |\langle p, n, n' \rangle|$. Because the terrain around n is uniform, the moves $\langle n, n' \rangle$ and $\langle p, p' \rangle$ have the same cost, as well as the moves $\langle p, n \rangle$ and $\langle p', n' \rangle$. It is then clear that A is shorter than B, and therefore the move $\langle n, n' \rangle$ is not optimal and can be pruned. □

Prospective g Pruning While diagonal branch pruning will avoid all of the horizontal scans shown in Figure 3, it will not prevent the vertical overscanning shown. The basic neighbourhood-based pruning rules are not powerful enough to determine that the vertical moves are unnecessary. *Prospective g pruning* utilises the g-values produced by the search to prove that these moves are unnecessary, allowing diagonal branch pruning to the eliminate the scans.

We maintain for each cell a *prospective g-value* (denoted pg), separate from the search g-value, as well as a flag indicating if the cell was reached through a prospective orthogonal move. When the search reaches a node n with a lower g value, we attempt to lower the prospective g-values of each of its neighbourhood successors according to Algorithm 1.

When the search expands a node n, we prune any move $\vec{m}$ to a cell x which has a prospective g-value less than the length of the path to n plus the cost of the move $\vec{m}$. Additionally, $\vec{m}$ is pruned if it is a diagonal move, x is flagged as being reached through a prospective orthogonal move, and the cost is the same as the prospective g value.

Theorem 3 *Prospective g pruning never removes an orthogonal last optimal path.*

Proof: The proof is straightforward. Let $\vec{m}$ be a move from node n to node s which is removed by prospective g prun-

Algorithm 1 Prospective g update

Require: n is a node reached in direction $\vec{m}$.
for $s \in$ neighbourhood successors of n **do**
  $pg \leftarrow n.g + |\langle n, s\rangle|$
  if $pg < s.pg$ **then**
    $s.pg \leftarrow pg$
    $s.ortho \leftarrow \langle n, s\rangle$ is orthogonal
  else if $pg = s.pg$ **then**
    $s.ortho \leftarrow s.ortho$ **or** $\langle n, s\rangle$ is orthogonal
  end if
end for

ing. Then the move $\vec{m}$ results in a path with tiebroken cost strictly greater than the tiebroken cost of some other path which set the prospective g value. The move $\vec{m}$ is therefore not minimum tiebroken cost, and can be pruned. □

Example 2 Consider the the example shown in Figure 4. When p is reached its neighbours are marked with a prospective g value. In particular, assuming n_0 is the start of the search, cell n, the down right neighbour of p, is marked with prospective g value $7/2\sqrt{2}$, the path length of $\langle n_0, p, n\rangle$.

Now when p' is expanded we examine the cost to n via p' which is $3 + 3/2\sqrt{2}$. Since this is greater than the marked prospective g value $7/2\sqrt{2}$, we prune the down move successor. This avoids the vertical overscan in the third column.

Using diagonal branch pruning, since p' has no down successor, we do not consider creating down successors for any successors of p' generated by down right moves. The combination of prospective g pruning and diagonal branch pruning eliminates all overscanning shown in Figure 3. □

Orthogonal Jump Caching The combination of prospective g and diagonal branch pruning allows all of the overscan in Figure 3 to be eliminated. However, there still exist common patterns that produce a significant amount of overscan that these methods are unable to prune. This overscan is produced by distant diagonal scans generating overlapping orthogonal scans. The stripes map Figure 5 is an example of such a map, since scans leaving the diagonal at different heights are more than one cell apart the prospective g pruning does not prevent the overscan.

To improve performance in this case, we introduce a caching scheme for orthogonal jumps which allows us to skip to the end of a jump if it has already been computed as part of a previous scan. Each cell maintains the length and cost of a jump in each orthogonal direction. The jumping procedure is modified to use this cache to complete scans early and to fill in cache values afterwards. The new jumping procedure is given in Algorithm 2.

Using jump caching we can guarantee that no cell is scanned more than 4 times (one for each orthogonal direction), reducing worst-case time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. There is a cost though, the scanning method is slower since we need to examine each cell for a cached value during a jump and fill in the new cache values after a jump.

Cache Maintenance Values in the jump cache are valid while the weighted grid map does not change. This means

Algorithm 2 Caching Jump Procedure

Require: n is a node and $\vec{d}$ is an orthogonal direction.
$cost \leftarrow 0$
$distance \leftarrow 0$
$steps \leftarrow 0$
repeat
  if $n(\vec{d}).cached$ exists **then**
    $cost \leftarrow n(\vec{d}).cached.cost$
    $distance \leftarrow n(\vec{d}).cached.distance$
    break
  end if
  $n \leftarrow n + \vec{d}$
  $steps \leftarrow steps + 1$
until cannot jump through n
while $steps > 0$ **do**
  $p \leftarrow n - \vec{d}$
  $cost \leftarrow cost + |\langle p, n\rangle|$
  $distance \leftarrow distance + 1$
  $n(\vec{d}).cached.cost \leftarrow cost$
  $n(\vec{d}).cached.distance \leftarrow distance$
  $steps \leftarrow steps - 1$
  $n \leftarrow p$
end while
return $cost, distance$

that we can safely reuse the cache for separate path finding queries until the map changes. This can substantially speed up multiple path finding calls while the map remains static. Of course, one of the principle advantages of jump point search is the ability to handle dynamically changing maps without recomputation.

A simple option to keep the cache in a valid state is to empty the cache when the map changes at all. This is guaranteed correct, and still ensure that we do not scan a cell more than 4 times in any single path planning call.

A better option is when we modify the map at locations $(x_i, y_i) \in L$ to remove cached values for every cell (x, y) which is within one row or column of a cell in L, i.e. $\exists i.|x - x_i| \leq 1 \vee |y - y_i| \leq 1$. Remember that orthogonal jumping depends also on terrain in adjacent rows or columns. If the number of map changes between path finding calls are small this can be much more efficient than simply emptying the cache. If L becomes too large, a simple emptying of the cache is probably more efficient.

Experimental Setup

For our experiments, we implemented JPSW in C++ using the warthog pathfinding research library. Full code is available at `bitbucket.org/dharabor/pathfinding`. The experiments were run on an AMD Ryzen 9 5950X clocked at 4.6GHz with 16GB 3200MHz DDR4 memory. Each experiment computes the average total time to answer a set of queries on each map in the benchmark set over five runs. The results are displayed as the speedup of the method with respect to using standard A* search, which is the current state of the art approach to path planning on weighted

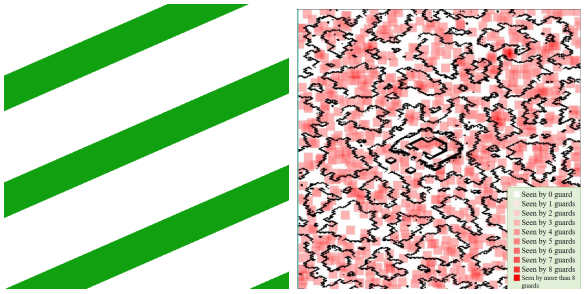

Figure 5: Left: a sample map from the Stripes set, using an angle of 24° with light region of width 128 and heavy region of width 64. Right: a map with 1024 Guards added.

terrain grid maps without using precomputation.

We performed four sets of experiments to test the effectiveness of JPSW in various scenarios:

Stripes maps: 36 synthetic two-terrain maps of size 512×512 that represent the worst cases for overscan. The terrain is arranged in stripes of various combinations of widths (64, 128, and 256) and at various angles (0°, 5°, 24°, and 45°). We generated 1000 random queries for each map in this set. An example map is presented in Figure 5.

Cities maps: 30 two-terrain maps of size 512×512 discretised from building and road maps available at Moving AI (Sturtevant 2012). We assign various costs to the normally impassable `@` terrain for these experiments. If the cost is less than 1 then paths try to shortcut through buildings, while if greater than 1 they will only enter buildings to avoid longer paths on the roads.

Multi-Terrain maps: We have two sets of maps with multiple terrain types. The Island set from (Sturtevant et al. 2019) contains 20 synthetic island maps of size 2048×2048 with approximately 15 terrain types, each assigned a random cost between 1 and 5. The WC3 set contains 36 maps from WarCraft 3 scaled up to 512×512 and has five terrain types, including impassable terrain. We consider trees to have cost 1.5, swamp cost 2, and water cost 4. Both of these sets of maps are available at Moving AI.

Guards: This set of maps is designed to test the sensitivity of JPSW to symmetry. Consider a scenario where an agent wants to go from a start node to a target node, and the map is filled with enemy Guards. Each Guard has a range where they may spot the agent. We model detection risk through terrain cost, increasing by 1 for every guard in range of the cell. We generated six maps of increasing complexity by adding various numbers of Guards (0, 64, 128, 256, 512, 1024) to the base map. The base map was *The Frozen Sea* from StarCraft, available at Moving AI (See Figure 5).

Methods Compared

In the experiments we consider a number of variants of weighted jump points search and their combination:

Base JPSW Using only the neighbourhood successor and jumping rules. Common to all variants.

Pruning Adding diagonal branch pruning and prospective g pruning.

Empty Cache Using jump caching but emptying the cache between each path planning query.

Full Cache The caches are pre-filled and never cleaned (the map is static).

For simplicity of comparison, all variants have the neighbourhood successor cache pre-filled. The overhead of filling the cache during search is small, between 10 and 30% depending on the map. Filling this cache when the map is loaded takes on average 0.28μs per tile on our test system.

The Full Cache benchmarks are presented as an upper bound on what JPSW is able to achieve. In practice, where the neighbourhood successor and jump caches are not pre-filled and multiple queries are run, performance will start out the same as Empty Cache and trend towards Full Cache over time. In dynamic environments, the frequent invalidation of cache entries may prevent performance from reaching that of Full Cache, although due to the invalidation strategy will still perform better than Empty Cache.

Results

Experiment 1: Stripes This experiment illustrates why Base JPSW is not sufficient, since we can see from Figure 6 that it actually runs slower than the baseline A* search on these maps, since overscanning is so common. For the horizontal 0° stripes Pruning is able to remove all overscanning (analogous to Figure 3), so immediately results in a significant speedup. Cache causes scanning to slow down, and hence is not as effective as Pruning on these maps. Once we move to slanted stripes Pruning is less effective, however it continues to provide benefits when used in conjunction with a cache. The best combination, Full Cache + Pruning avoids most redundant scans and for later queries can reuse the cache to avoid almost all scanning leading to massive improvements.

Experiment 2: City Maps In the results shown in Figure 6 we see that Base JPSW is worse than A* when the buildings are lighter terrain than the roads. Adding Pruning makes JPSW always better than the baseline. In these maps where scan distances are not so great, Empty Cache is always beneficial for the lighter building terrain, while the costlier scanning does not always pay off for heavier terrain. Full Cache is always beneficial. Overall JPSW leads to a more than five time improvement over A* regardless of terrain cost. With infinite cost this is equivalent to the original single terrain map where Pruning is simply overhead, but Full Cache does lead to significant speedups.

Experiment 3: Multi-Terrain maps The results show that Base JPSW causes slowdown on the DWA maps because of the many terrain types which lead to frequent overscan, while for WC3 it is slightly better than A*. Adding the Pruning or Empty Cache techniques are both enough to beat the baseline A*, and the best combination Full Cache + Pruning leads to significant speedups of 8 and 4.5 times.

Experiment 4: Guards Results are shown in Figure 8. The 0 guard case is simply a single terrain map, and hence overscanning is rare and Pruning is simply overhead. As we add more guards we reduce the symmetry available to take

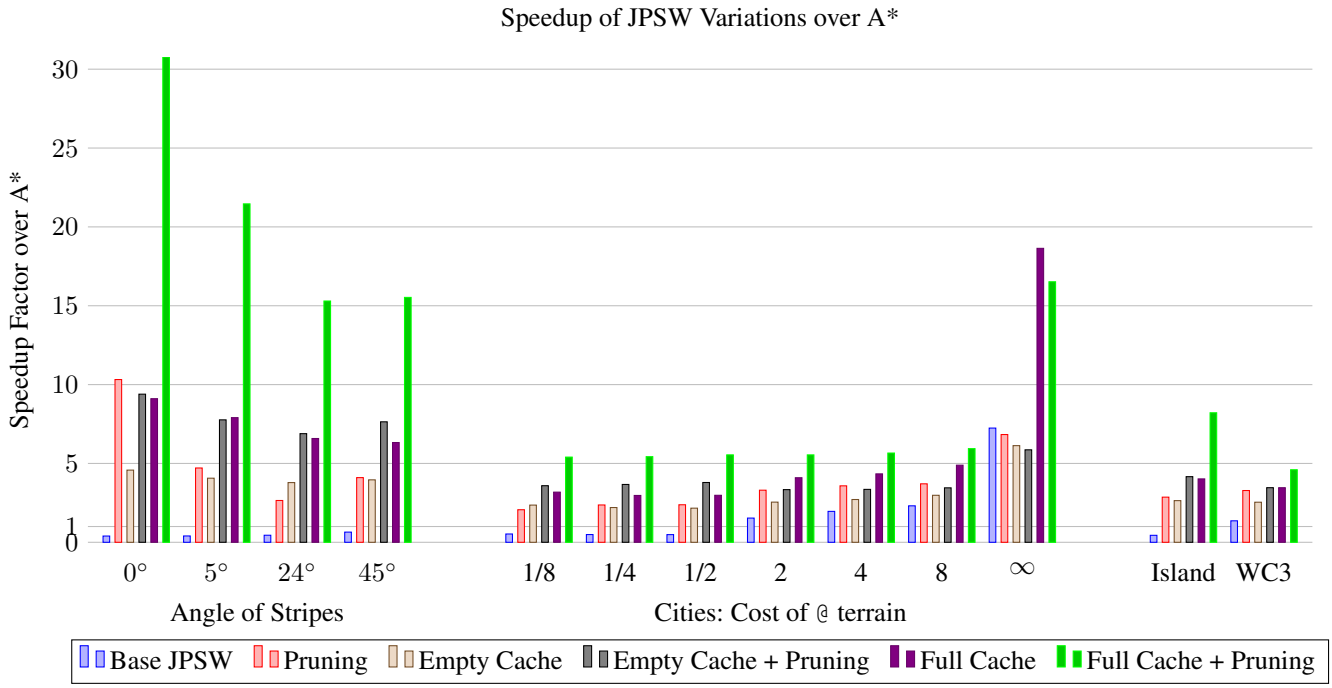

Figure 6: Speedups over A* for the Stripes, City Maps and Multi-terrain Maps for the various configurations

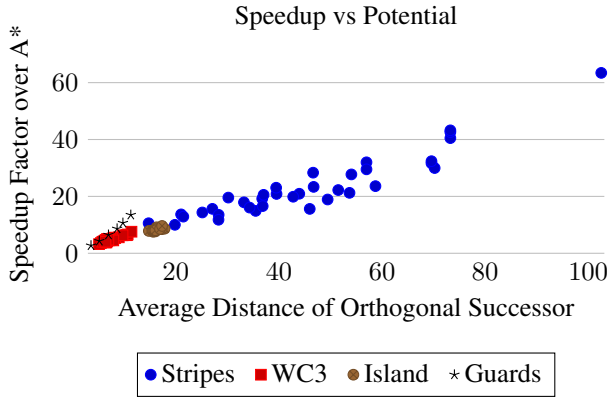

Figure 7: Speedup factor of Pruning + Full Cache over A* versus a measure of the potential speedup of the method on the map.

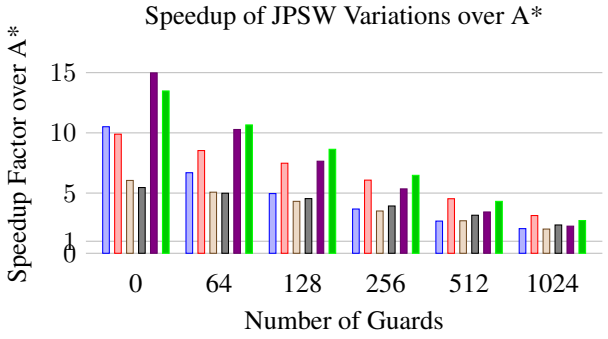

Figure 8: Speedups over A* for the various configurations on the Guards scenarios.

advantage of and all of the JPSW methods become increasingly similar to A*, however none end up slower over all. Pruning is the most important improvement, while Cache methods are less effective because the jump distances become increasingly short as we add more guards.

Variance in speedup In Figure 7 we measure the relationship between speedup and the average jump distance to each orthogonal successor, across all our tested maps. The strong correlation indicates JPSW primarily achieves its speedup by skipping nodes in locally uniform regions, most importantly during orthogonal scans,where we can use the jump cache to skip directly to the end of the jump.

Conclusion

We introduce Weighted Jump Point Search (JPSW), a fast and optimal equivalence-breaking approach for path planning on weighted grid maps, of the kind that often appear in computer games. We show that breaking equivalences is much harder in this domain than in the well known uniform-cost case. Our new pruning techniques can nevertheless effectively reduce the size of the search space, and the number of operations on the OPEN list, all while guaranteeing solutions remain grid-optimal. In a range of experimental settings, drawn from or inspired by computer games, we report speedups of up to one order of magnitude, on average, vs the most competitive baseline in the area, online A* search. Future work includes more sophisticated jump stopping conditions, and stronger pruning rules to reduce overscan.

Acknowledgements

This work was partially supported by Australian Research Council Grant DP200100025.

References

Bono, M.; Gerevini, A. E.; Harabor, D. D.; and Stuckey, P. J. 2019. Path Planning with CPD Heuristics. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 1199–1205. International Joint Conferences on Artificial Intelligence Organization.

Cohen, L.; Uras, T.; Jahangiri, S.; Arunasalam, A.; Koenig, S.; and Kumar, T. 2017. The FastMap algorithm for shortest path computations. *arXiv preprint arXiv:1706.02792*.

De Carufel, J.-L.; Grimm, C.; Maheshwari, A.; Owen, M.; and Smid, M. 2014. A note on the unsolvability of the weighted region shortest path problem. *Computational Geometry*, 47(7): 724–727.

Delling, D.; Goldberg, A. V.; Savchenko, R.; and Werneck, R. F. 2014. Hub labels: Theory and practice. In *International Symposium on Experimental Algorithms*, 259–270. Springer.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, 319–333. Springer.

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. D.; and Grastien, A. 2012. The JPS Pathfinding System. In *SOCS*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Mitchell, J. S.; and Papadimitriou, C. H. 1991. The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM (JACM)*, 38(1): 18–73.

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.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Sturtevant, N. R.; Sigurdson, D.; Taylor, B.; and Gibson, T. 2019. Pathfinding and abstraction with dynamic terrain costs. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 80–86.

Tran, N.; Dinneen, M. J.; and Linz, S. 2020. Computing Close to Optimal Weighted Shortest Paths in Practice. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 291–299.