Proceedings of the Sixteenth International Symposium on Combinatorial Search (SoCS 2023)

Reducing Redundant Work in Jump Point Search

Shizhe Zhao, Daniel Harabor, Peter J. Stuckey
Monash University
{shizhe.zhao,daniel.harabor,peter.stuckey}@monash.edu

Abstract

JPS (Jump Point Search) is a state-of-the-art optimal algorithm for online grid-based pathfinding. Widely used in games and other navigation scenarios, JPS nevertheless can exhibit pathological behaviours which are not well studied: (i) it may repeatedly scan the same area of the map to find successors; (ii) it may generate and expand suboptimal search nodes. In this work, we examine the source of these pathological behaviours, show how they can occur in practice, and propose a purely online approach, called *Constrained JPS* (CJPS), to tackle them efficiently. Experimental results show that CJPS has low overheads and is often faster than JPS in dynamically changing grid environments: by up to 7x in large game maps and up to 14x in pathological scenarios.

Introduction

Grid Based Pathfinding is a classic problem in AI and widely used in games and robotics, as well as an active research area. Despite a variety of fast preprocessing-based approaches for solving this problem (Sturtevant et al. 2015), online approaches are still preferable in dynamic environments where *obstacles may appear (or disappear) on the map between different queries*. The reason is that preprocessing-based approaches rely on precomputed auxiliary data structures, which have to be rebuilt or repaired when the map changes. When dynamic changes are frequent and/or affect large regions of the map, the online costs of rebuild or repair operations grows quickly and the amortized performance of preprocessing-based algorithms becomes worse than purely online approaches (Mahéo et al. 2021; Hechenberger et al. 2020).

Jump Point Search (JPS) (Harabor and Grastien 2011, 2014) is a state-of-the-art algorithm for online pathfinding on uniform-cost grids. It applies a local pruning policy to prune the successor generation of A* to reduce symmetries. Experiments show that JPS is two orders of magnitude faster than A*. However, it may suffer from some pathological behaviours. These behaviours cause it to be inefficient in certain topologies of maps that are common in dynamic environments. In further sections, we will provide technical details of JPS, and discuss these pathological behaviours.

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this work we examine the root cause of JPS's pathological behaviours. We then consider how to resolve these issues on-the-fly using a new reasoning technique based on geometric constraints. This approach reduces redundant work and helps to avoid the generation and expansion of suboptimal successors. We then integrate these constraints with block-based scanning to produce a new state-of-the-art algorithm called *Constrained JPS* (CJPS). Finally we conduct an experimental comparison on benchmarks drawn from real applications and on synthetic setups intended to produce pathological behaviour. Results show that when the environment is dynamic CJPS can be up to 7x faster than JPS in terms of cumulative runtime. In pathological setups, this improvement can be up to 14x. When there is little or no redundant work, CJPS has a small overhead of $\approx 25\%$. Overall, *Constrained JPS* represents a substantial advancement for optimal and online grid-based pathfinding in dynamic environments.

Background

In this section we give a brief description of the dynamic grid-based pathfinding problem. We then review related works for dynamic environments, to define our research scope. We also give a summary of the JPS algorithm, to help readers understand the weaknesses of the current approach.

Problem Definition

The gridmaps we consider in this paper are 8-connected, with move directions: *N,S,W,E,NW,NE,SW,SE*. Each cell in the map is either traversable or blocked. Our vertices $\mathbf{V}$ are located at the coordinates of the traversable cells. Edges $\mathbf{E}$ are defined from vertex to adjacent vertex using one of the move directions. **Corner-cut** is not allowed, i.e. in a valid diagonal move, adjacent cells in both of the component cardinal move directions (which together comprise the diagonal vector), must also be traversable; e.g., in Figure 1b, the agent cannot move from 4 to 2 as the north cell 1 is blocked. For $e \in \mathbf{E}$, let $\mathbf{w(e)}$ be the cost of each move, all **cardinal moves** (*N,S,W,E*) have cost 1, and all **diagonal moves** (*NW,NE,SW,SE*) have cost $\sqrt{2}$. A **path** is represented by $p = [v_0, v_1, \ldots, v_k]$, where $v_i \in \mathbf{V}$ and $(v_i, v_{i+1}) \in \mathbf{E}, 0 \leq i < k$. The length of a path p is $len(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$. The shortest path from s to t is

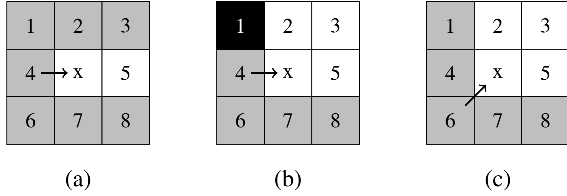


Figure 1: JPS pruning example. In all figures: x is the current node, arrow shows the incoming direction, black cells are blocked, white cells are successors and gray cells are pruned.

a path with the minimum length. The octile distance between vertices a, b is denoted by $|ab|$. We assume the environment can change between pathfinding queries. We further assume that these changes are uniformly distributed on the map.

Related Works for Dynamic Environment

Pre-computed estimators, such as Landmark Heuristics (Goldberg and Harrelson 2005), Differential Heuristics (DH) (Sturtevant et al. 2009) and CPD Heuristics (Bono et al. 2019), have the potential to handle some types of dynamic environments. In particular, these approaches remain admissible under the assumption that dynamic changes online will never decrease the cost of any optimal path to less than its cost during the offline phase. Pre-computed heuristics are orthogonal to JPS, which does not come with any cost-based assumptions. Another approach, Customizable Contraction Hierarchies (CCH) (Delling et al. 2017) is an abstraction-based preprocessing technique which can repair its auxiliary data online, after a dynamic change. CCH is fast, optimal and it does not rely on any cost-based assumptions. A main disadvantage is that as the number and frequency of map changes increases the amortized cost of repair grows large, to the point where it becomes faster to use a reference algorithm such as A* (Mahéo et al. 2021).

Jump Point Search

Symmetry breaking is a main challenge in 2D path finding. This problem occurs when there exist multiple shortest paths from start to target, each of which is derived from any of the others by simply re-ordering grid moves. JPS eliminates symmetries by exploring only *canonical paths*, where diagonal moves appear as early as possible. The search framework of JPS is the same as A* and uses the same heuristics for prioritising nodes in the OPEN list. In this work we use **octile distance**, a popular heuristic similar to the Manhattan estimator but which allows diagonal moves. The main difference between JPS and A* is the successor function. In JPS, *local* suboptimal and non-canonical grid neighbours are pruned on the fly using a series of simple rules which are recursively applied. We define these rules and basic concepts of JPS below.

Definition 1. Let $NB(x) = \{v \mid (x, v) \in \mathbf{E}\}$ be the adjacent vertices of a current node x, and let p be the parent of x during search. Define $LP_p^x(t)$ as the set of optimal paths starting at p, ending at t, and restricted to only use vertices from $NB(x)$. We say that t belongs to the successor set of x (coming from p) if $\forall l \in LP_p^x(t)$:

$$len([p,x,t]) = len(l) \land rank([p,x,t]) \leq rank(l) \quad (1)$$

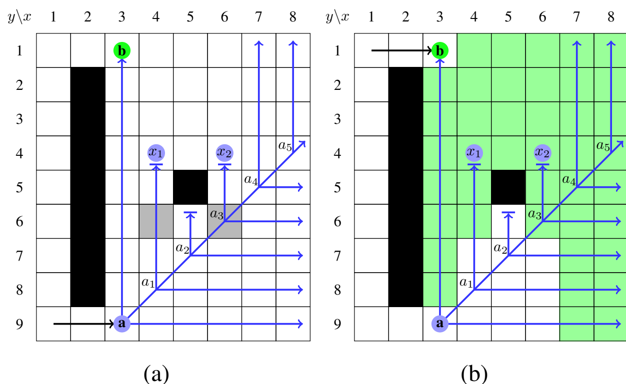
Where $rank(\pi)$ is a function that returns the index of the first diagonal move of a path π.

Example 1 In Figure 1a, let x be the current search node that comes from node 4. $LP_4^x(2) = \{[4,2]\}$, and $[4,x,2] \notin LP_4^x(2)$, so 2 is not a successor. $LP_4^x(3) = [4,2,3], [4,x,3], [4,x,2] \in LP_4^x(3)$ but the diagonal move in $[4,2,3]$ appears earlier than the diagonal move in $[4,x,3]$, i.e., $rank([4,x,3]) > rank([4,2,3])$, so 3 is not a successor.

Concepts: we refer to the successor set of node x with the notation $neib(\vec{d}, x)$, where $\vec{d}$ is the incoming direction inferred from p and x. These are sometimes called the **diagonal-first** (equiv. canonical) neighbours of node x and they can be computed on-the-fly in constant time. When moving in a straight direction $\vec{d}$, JPS often reduces the pruned successor set to size 1, i.e., $|neib(\vec{d}, x)| = 1$ (see Fig 1a). Rather than adding such nodes to the OPEN list JPS immediately expands them, thus applying the pruning rules anew in a recursive fashion. The recursion stops when reaching a node x' where $|neib(\vec{d}, x')| = 0$ (**dead-end**) or when $|neib(\vec{d}, x')| > 1$ (**jump point**), which can occur due to adjacent obstacles (see Fig 1b). Recursing across the grid looking for jump point successors is called **scanning**. In the second case x' is generated as a **jump point** successor of x. A vertex v is a **corner point** if $\exists d \in \{N, E, W, S\}$: $|neib(\vec{d}, v)| > 1$. When moving in a diagonal direction $\vec{d'}$, JPS first scans in each of the two corresponding cardinal directions, looking for jump points. These are immediately added to the OPEN list and then JPS takes one diagonal step further on the grid (see Fig1c). This procedure is called **diagonal recursion** and it continues until the next move $\vec{d'}$ is a dead-end or corner-cut.

Example 2 In Figure 2a, node a is a jump point with successors (diagonal first neighbour set) {N, E, NE}. JPS first scans in directions N and E, and finds a jump point b to the north. Then it moves one diagonal step to a_1 and applies the same scanning, finding x_1 at the node above a_1 during the North scan. Similarly, on the North scan, JPS will find a dead-end from a_2, and a jump point from a_3. Grey cells are corner points that wouldn't stop the scanning. JPS keeps moving diagonally and scanning cardinally until the diagonal direction is blocked.

Block-Based Scanning. *Block-based scanning* (Harabor and Grastien 2014) represents the traversability of the gridmap by a bitmap, i.e. 0/1 means traversable/non-traversable; then instead of checking node by node, we can load a block of bits for adjacent columns/rows into memory, and quickly compute the position of the first jump point in this direction if there is one, by using three bitwise operations. This procedure is branch-less and leverages SIMD instructions which can be extremely fast, it improves average times of JPS by nearly one order of magnitude. The only



(a) (b)

Figure 2: In (a), node a is the current search node, the black arrow represents the coming direction, blue arrows represent scanning, $a_{i\in 1\dots 5}$ are start points of each cardinal scanning in diagonal, gray nodes are corner points, b, x_1, x_2 are identified successors (jump points). In (b), b is another search node that performs a symmetric diagonal recursion, and green nodes are scanned during this procedure.

overhead is we need two bitmaps, one stored rowwise and one stored columnwise. For more details see example 3.

Example 3 Assuming the block size is 3, in Figure 1b, the bit string of each row ('1,2,3','4,x,5' and '6,7,8') are $s_1 = 100_2$, $s_2 = 000_2$, $s_3 = 000_2$. In a left-right travel, we can quickly compute the first position that bits change from 1 to 0 by bitwise operation on s_1, which indicates a traversable node next to an obstacle; apply same reasoning on s_2 and s_3, we will find that x is a jump point.

Pathological Behaviours in JPS

Known for its fast performance and strong optimality guarantees, JPS nevertheless suffers from two distinct types pathological behaviour, each of which can create substantial amounts of redundant work.

Pathological Behaviour #1: JPS may scan the entire map multiple times while computing successors during search. In (Sturtevant and Rabin 2016) the authors tackle this issue by adding a scan limit during the successor generation step of JPS. The resulting algorithm, known as Bounded JPS, introduces a successor node each time the limit is reached. This approach mitigates redundant scanning but at the cost of additional heap operations on the OPEN list.

Moreover practitioners must carefully choose a suitable limit, usually for each map and potentially for each query, in order to achieve a performance improvement. A related algorithm, Boundary Lookup JPS (Traish, Tulip, and Moore 2016), replaces the linear scanning operation of JPS with a binary search. This approach has better complexity but is much slower in practice than optimised JPS implementations, which rely on an instruction-level parallelism technique known as block-based scanning (Harabor and Grastien 2014).

Pathological Behaviour #2: JPS may generate and expand suboptimal search nodes. This behaviour has not been reported previously in the literature but could be a cause for concern — if many suboptimal search nodes are created and expanded, there is a potentially large overhead for JPS vs. A*, since A* only expands each node at most once (assuming with a consistent heuristic). It is therefore important to understand how suboptimal expansions can occur in JPS, how much performance is affected by this behaviour and how to avoid it in practice.

Example 4 In Figure 2b, expanding nodes a and b produces overlapping diagonal recursions where nodes are scanned multiple times. Depending on the g-value of a and b, some of the resulting successor nodes may be suboptimal. For example, when $g_a = g_b$, x_1, x_2 are bettered reached from b, but JPS still generates them as successors of a. Furthermore, suboptimal nodes x_1, x_2 will be expanded later.

Constrained JPS

Observation In Example 4, scan from a will stop at b (as it is a jump point to a), and the same applies to b. Therefore, a and b have a chance to know each other's g-value during their diagonal recursion, which can be utilized to reduce redundant work.

Based on the observation above, we propose *Constrained JPS*. At a high level, when we expand node a and the cardinal scanning finds a jump point v with an existing g value, we can create a constraint for the same cardinal direction during the current diagonal recursion. This constraint restricts cardinal scanning in the later diagonal recursion by a dynamic jump limit, and it can be deleted or updated during the recursion. We first give a formal definition of the constraint and then we demonstrate how to compute it, when to delete or update it and how to use it for pruning.

Definition 2. A *constraint* is defined by a tuple $\langle a, v, \vec{d}, L\rangle$, where:

- a is a vertex which starts or continues a diagonal recursion;
- $\vec{d}$ is the cardinal scanning direction in the diagonal recursion;
- v is a jump point found by a scanning in direction $\vec{d}$ from a, which has an associated g-value (i.e., it has already been included in the OPEN list or has been expanded);
- L is an integer that indicates the maximum number of recursive diagonal moves from a. A constraint is **applicable** if the number of diagonal moves from a to a_i is not greater than L.

For the remainder of this section, we use i for the i-th diagonal move from a and a_i for the corresponding vertex.

L guarantees that when a constraint is applicable, the cardinal scanning in direction $\vec{d}$ on a_i shouldn't take more than $|av| - i$ steps. Such a restriction is a perpendicular blockage for scanning in direction $\vec{d}$ and any path crossing the blockage is better reached from v. Figure 3a illustrates the idea. We now show how to compute L.

Computing L Let g_a be the g-value of node a and g_v be the g-value of node v found when expanding a. To compute L, we need to consider the following cases:

(i) If $g_a + |av| \leq g_v$, all scans during the diagonal recursion at a will cross the blockage with a better g-value than from node v; no constraint is applicable, so we set $L = 0$.

(ii) If $g_a + \sqrt{2}|av| > g_v + |av|$, the diagonal recursion from a should be terminated after $i = |av|$ diagonal steps; since all further nodes are better reached from v we let $L = |av|$;

(iii) Otherwise L is the minimum integer satisfying:

$$\begin{cases} L \geq 0 \wedge L \leq |av| \wedge \\ \overbrace{g_a + \sqrt{2}L + (|av| - L)}^{\text{distance from a}} < \underbrace{g_v + L}_{\text{distance from v}} \end{cases}$$

Updating and Deleting Constraint When the constraint is applicable, at the i-th diagonal step (i.e., at a_i), there are two cases in the cardinal scanning: i) the scan stops at the blockage; ii) the scan is stopped by a jump point or a dead end at node p, before reaching the blockage. In the first case, we can still use the constraint as long as it is applicable. In the second case, we must discard the current constraint and create a new constraint on p. To do this, we need to estimate a tight upper bound for the g-value of p:

$$\overline{g}_p = \min\{g_p, |vp'| + 1\} \quad (2)$$

where g_p is the g value currently stored with p (or $+\infty$ if it has no stored value), and p' is the cell in the previously scanned row/column adjacent to p. This bound is safe since the previous scans of rows/columns from v to p' must be empty up to the blockage, since we haven't yet deleted the constraint from v. Therefore, we update the constraint to $\langle a_i, p, d, L' \rangle$, where L' is computed (as discussed previously) based on $g_{a_i}, \overline{g}_p$ and $|a_i p|$, Figure 3b gives an example. When $L' = 0$, the new constraint is not applicable, then we can simply delete it.

Pruning and Early Termination Equation 2 can estimate whether node p is better reached from v than a. When p is a jump point, we can prune it if:

$$\overline{g}_p < g_{a_i} + |a_i p|$$

and when p is at the blockage, we can terminate the diagonal recursion early if:

$$\overline{g}_p + |a_i p| < g_{a_i}$$

Algorithm 1 illustrates the horizontal constraint of CJPS, the vertical constraint is imposed similarly. In practice, both can be applied orthogonally in the same diagonal recursion.

Branch-less Implementation

Block-based scanning can improve JPS by nearly an order of magnitude (Harabor and Grastien 2014), thus we need the proposed approach to work with it. CJPS computes a jump limit on-the-fly based on the information obtained from scanning. However, blocked-based scanning is branch-less leveraging SIMD instructions, so applying a jump limit

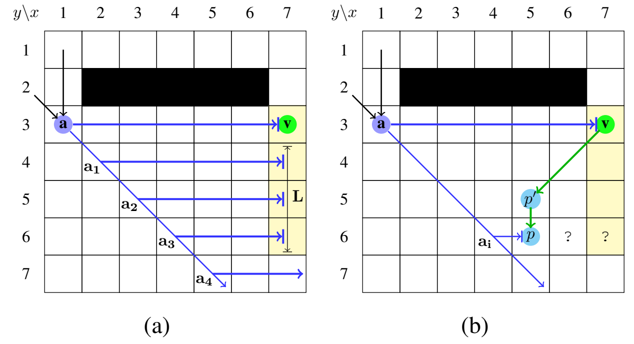


Figure 3: (a) shows constraint (a, v, d, L), where a is a jump point coming from north, or a continued diagonal move from northwest; v is a previously visited node that found from the cardinal scanning from a, and L is the maximum number of steps where the constraint is applicable, yellow cells are better reached from v, thus cardinal scanning from a_1, a_2, a_3 is stopped by the constraint at the yellow blockage; (b) shows how to evaluate $\overline{g}_p$, where p is the stop location of a scanning within the constraint. The traversability of all nodes after p is unknown, denoted by '?'.

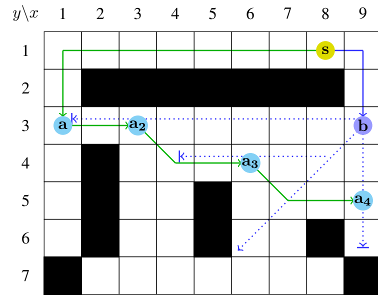


Figure 4: s (yellow) is the source node, a (cyan) and b (blue) are successors of s. a_2, a_3, a_4 are suboptimal nodes propagated from a, their optimal parent is b, but the scanning from b won't stop at them.

is not trivial since adding any if-then-else statement would significantly slow down the scanning. To force the scanning to stop at certain jump limit in a branch-less way, instead we set an artificial obstacle on the map before the scanning and unset it afterwards.

Eliminating Suboptimal Node Expansion

We have shown that *CJPS* can prune redundant scanning and suboptimal nodes, but it doesn't guarantee to eliminate all of them. Other online algorithms, e.g. A* and Dijkstra, are less efficient in practice but guarantee no suboptimal node expansion. Thus we need to answer the following questions: 1) how does this happen? 2) how bad could it be? 3) can we eliminate all suboptimal node expansion?

Algorithm 1: Horizontal constraint in diagonal recursion.

Input: $n = (x, y)$: the location of current search node;
$d = (dx, dy)$: the direction of the diagonal move;
$d_h = (dx, 0)$: the direction of horizontal scanning;
$c_h = (a, v, d_h, L)$: the constraint for horizontal scanning;
db: the bit map for block based scanning;
`calcL(`$g_a, g_v, |av|$`)` : computing L based on definition 2;
`calcG(`v, p`)` : computing upper bound of g_p based on equation 2;
`scan(`n, d, JL, db`)` : blocked based scanning at n in d with the jump limit JL;

```
1  i ← 0;
2  while empty(x+dx, y) and empty(x, y+dy) do
3      n ← n + d;
4      if i ≤ L then
5          i ← i + 1;
6      if i ≤ L and n is better reached from v then
7          break;
8      if i ≤ L then
9          JL ← |av| − i;
10     else
11         JL ← ∞;
12     p ← scan(n, d_h, JL, db);
13     if |np| < JL then
14         g_p ← calcG(v, p);
15         c_h ← n, p, d_h, calcL(g_n, g_p, |np|);
16         i ← 0;
17     if p is jump point and better reached from n then
18         successors.add(p);
```

How Does It Happen?

Let a be a non-target search node that is expanded by JPS with suboptimal g-value. We can infer that node a must be a corner point (cf. jump point) when scanning from its optimal parent – call this optimal direction $\vec{d}$. In other words $(\vec{d}, n)$ does not form not a jump point from the optimal parent. Also, since node n is not the target node either, it cannot be generated as a successor when we expanded its optimal parent. Notice that the optimal path to the target remains in the JPS search space, but the optimal path to n does not. Figure 4 shows an example.

We now see that expanding a suboptimal node *v* requires two conditions: (i) *v* is reachable from at least two parents (i.e., jump points or the start node); (ii) *v* does not appear in the search space of its optimal parent due to the JPS pruning rule in Definition 1. This situation is more likely to happen when the map has many corner points and when the heuristic is less accurate; e.g., on game maps.

How Bad It Could Be?

The successors of a suboptimal search node may also be expanded, and the suboptimality may be propagated; e.g. nodes a_2, a_3, a_4 in Figure 4. When this happens it is possible that suboptimal node expansions can dominate the entire search. It is hard to perform a theoretical analysis on the number of suboptimal node expansions in general because such behavior depends on many factors, such as the topology of the map, the order of expansion, and the target location. To understand how frequently this happens in practice we solve a subset of queries for a variety of benchmark domains. We counted the number of suboptimal node expansions per query and we also counted the number of propagations; i.e., the number of expanded nodes whose parent is suboptimal. Table 1 shows the result. We see that, there are no suboptimal nodes in *maze512* domain. This is due to the underlying tree structure of these maps, which means that search nodes are reachable from only one jump point. In other domains, more than 25% of node expansions are suboptimal, and many of these are propagated from their parent. This implies that pruning suboptimal nodes earlier could avoid expanding more suboptimal nodes in the future.

domain	#maps	mean	median	sub%	subp%
dao	154	215	62	30	64
bgmaps	75	98	41	27	65
starcraft	75	1038	777	28	73
street	30	596	312	33	84
iron	35	5065	3962	43	88
maze512	6	8863	2516	0	—
random10	1	7574	7527	33	18
rooms	4	1766	653	25	26

Table 1: Statistic of each domain. We count node expansions for the 100 longest queries of each map. *mean, median* are for node expansions, which indicates the difficulty, *sub%* shows the proportion of suboptimal node expansions, and *subp%* shows the proportion of propagated suboptimal node expansion among *sub*.

Can We Eliminate All of Them?

Based on the discussion above, it is clear that we can eliminate all suboptimal node expansion by storing a g-value on every corner point, rather than just on every jump point. However, this approach comes with significant overhead since, for each cardinal scan, we needs to stop at each encountered corner point. Although this does not change the total number of scanned nodes, it can significantly affect the performance of block-based scanning. Therefore there is a trade-off, between introducing overhead on the one hand and reducing redundant work on the other. In this section we explore two ways to mitigate redundant expansions with minimal overhead costs.

Diagonal Caching Here we store a g-value on each corner point found during diagonal recursion. Since the diagonal recursion stops at every cell on the diagonal, there is little overhead, mainly arising from an extra memory access during the scan. However, this approach only allows pruning cells that appear along diagonals from an expanded node, and most cells do not meet this condition.

Backwards Scanning When a cardinal scan in direction $\vec{d}$ finds a jump point n, we start a new scan from n in the reverse direction. The forward to n can pass at most two

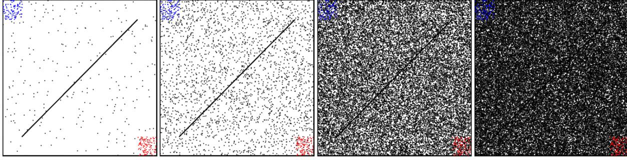


Figure 5: Synthetic maps, where $s = 512$, $b = 75\%$ and $r \in \{0.1, 1, 10, 20\}$. Blue and red clusters are starts and targets.

corner points before reaching n - one for the row above and row below (or the column left and right). The reverse scan can thus stop after labelling at most two corner points (or upon reaching the parent of n, whichever comes sooner). Since not all (forward) scanning finds a jump point (most are dead-end) this approach should have smaller overhead than stopping at all corner points. However it is more expensive than Diagonal Caching.

Experimental Setup

We compare *CJPS* and *JPS* on two distinct benchmarks: synthetic maps, which shows performance in extreme cases, and domain maps, which shows performance on a range of well-established test sets drawn from real applications. Table 1 shows a summary. CJPS and JPS are both implemented in C++ and both make use of block-based scanning (Harabor and Grastien 2014). We compile with *clang 13* using *-O3* under *5.10.102-1-MANJARO* and we test on a *Intel Xeon E-2276M* processor with *32 GB* RAM. Our implementations and data are available online.[1]

Domain maps: From Sturtevant's benchmark set (Sturtevant 2012) we select all game domains (bgmaps, starcraft, dao) and all grid rasterisations of real cities (streets). We also experiment with the Iron Harvest domain, a recent collection of 35 grid maps taken from the game (Harabor, Hechenberger, and Jahn 2022). These maps are much larger than and much more challenging than other game benchmarks.

Synthetic maps: these are pathological test cases featuring an empty map with random obstacles and a diagonal blockage in the middle. We control for three variables:

- r: the proportion of traversable cells that are blocked, which simulates dynamic environments;
- s: the height and width of map is $s \times s$;
- b: the proportion of diagonal blockage, which controls the difficulty, i.e., less accurate heuristic;

There are 100 instances per map, where starts and targets are always traversable and clustered in top-left and bottom-right regions. Figure 5 shows an example.

Results

For all experiments that measure execution time, we run each map 10 times in random order and choose the median, to avoid cache behaviour and reduce random noise. For all experiments that measure suboptimal behaviour, we run a Dijkstra to compute a true-distance table before each query.

[1] https://github.com/eggeek/constrained-jps

r (%)	Improvement Factors			
	hp-opt	subopt	expd cost	runtime
0.0	1.00	—	0.79 (5.76/ 7.34)	0.79
0.1	1.64	37.10	8.95 (3.24/ 0.36)	14.87
1.0	1.56	3.84	4.16 (1.51/ 0.36)	6.47
10.0	1.15	1.44	1.03 (0.39/ 0.37)	1.18
20.0	1.08	1.26	0.89 (0.30/ 0.34)	0.96

(a) Vary obstacle density, fix s=512, b=75%.

b (%)	Improvement Factors			
	hp-opt	subopt	expd cost	runtime
0.00	1.00	1.00	0.78 (12.00/ 15.19)	0.78
0.25	1.52	8.90	7.48 (5.02/ 0.66)	10.21
0.50	1.58	9.12	8.20 (3.81/ 0.46)	12.72
0.75	1.64	37.10	8.98 (3.26/ 0.37)	14.75

(b) Vary heuristic accuracy, fix r=0.1%, s=512

resolution	Improvement Factors		
	hp-opt	expd cost	runtime
256	1.31	2.85 (1.21/ 0.43)	3.80
512	1.29	4.62 (3.63/ 0.79)	5.97
1024	1.29	5.10 (11.32/ 2.24)	6.59
2048	1.29	5.47 (38.73/ 7.12)	7.12

(c) Vary resolution, scale up a map r=0.1%, s=256, b=75%

Table 2: Improvement factors of various metrics ($\frac{Metric(JPS)}{Metric(CJPS)}$) on three settings, >1 means improvements, where: (1) *hp-opt* measures heap operations (#expansion + #insertion); (2) *subopt* measures *opt* on suboptimal nodes; (3) *TPE* (time cost per expansion) measures average cost per node expansion in μs ($\frac{time}{\#expd}$), we also reveal the raw TPE of JPS and CJPS in parenthesis; (4) *runtime* measures average runtime per query ($\frac{time}{\#queries}$).

Exp-1: Synthetic Maps

This experiment shows how CJPS is affected by three map properties: random-obstacles density, heuristic accuracy and resolution (size of map). To do this, we run queries on three sets of synthetic maps:

- Fix s=512, b=75%, vary r $\in \{0, 0.1, 1, 10, 20\}$%;
- Fix r=0.1%, s=512, vary b $\in \{0, 25, 50, 75\}$%;
- Fix the map (r=0.1%, s=256, b=75%), vary resolution $\in \{256, 512, 1024, 2048\}$. This affects map size but not topology; i.e., the number of jump points is the same.

Diagonal Caching and *Backwards Scanning* **were not** applied in CJPS in this experiment. Table 2 shows the results. From Table 2a, we can see that when there is no chance to prune (r=0%), CJPS is 21% slower than JPS. This is due to the overhead of additional local reasoning in diagonal recursion. With 0.1% random obstacles pruning becomes effective and CJPS is 14.87 times faster than JPS. When r increases, the average expansion cost of CJPS is stable, and although there are more chances to prune (i.e., more search nodes), the improvement factor drops. The reason is

r(%)	jps	jps-g	jps-b	cjps	cjps-g	cjps-b
0.0	0.00	0.00	0.00	0.00	0.00	0.00
0.1	0.52	0.52	0.29	0.06	0.06	0.06
1.0	0.71	0.69	0.60	0.29	0.27	0.17
10.0	0.59	0.57	0.50	0.47	0.43	0.39
20.0	0.44	0.40	0.35	0.37	0.35	0.31

(a) Proportion of suboptimal expansion: $\frac{\sum SuboptExpd}{\sum Expd}$

r(%)	jps-g	jps-b	cjps	cjps-g	cjps-b
0.0	0.97	0.95	0.75	0.77	0.77
0.1	0.99	1.02	14.61	13.55	11.91
1.0	1.00	0.75	6.32	6.13	6.14
10.0	0.93	0.85	1.20	1.18	1.19
20.0	0.93	0.86	0.98	0.98	0.96

(b) Speedup factor: $\frac{avg(Time_{jps})}{avg(Time_{*})}$, where $*$ are variants: jps-g, jps-b, cjps, cjps-g, cjps-b.

Table 3: Results on synthetic maps. We look at suboptimal expansions and run time.

that the g-value differences between optimal and suboptimal nodes are smaller, and the upper bound estimation (equation 2) is relatively less accurate, which weakens the pruning (i.e., *subopt* drops). Meanwhile, JPS expansion cost becomes smaller as the diagonal recursion terminates earlier. Thus, CJPS becomes less effective and eventually slower than JPS when r=20%.

From Table 2b, we can see that CJPS is more effective when the heuristic is inaccurate, because the search space becomes larger and JPS tends to generate more suboptimal search nodes. In Table 2c we see the number of heap operations (*opt*) are the same on both maps. This is because simply scaling up doesn't change the number of jump points in the search space. We also see the expansion cost improvement of CJPS increases, especially up to 1024. The reason is that scanning a higher-resolution grid map is slower, due cache behaviour. Thus the reduced scanning in CJPS saves more time. As the map size grows to 2048, further improvement is diminished. Here CJPS has more cache misses, as the local reasoning needs to access entries in a larger g-value table (our table size scales with map size).

Discussion: CJPS can achieve significant improvements in dynamic scenarios (r>0%), especially when the heuristic is inaccurate or the resolution is high. Its improvement factor drops with the increasing density of obstacles.

Exp-2: Ablation Study

This experiment is to show whether it is worth further eliminate redundant work by diagonal caching (*-g*) and backwards scanning (*-b*). We run both variants ($\{$JPS, CJPS$\} \times \{$*-g, -b*$\}$) on the synthetic map set that shows CJPS is less effective when the density increases (s=512, b=75%, r $\in \{0, 0.1, 1, 10, 20\}$%). Table 3 shows the results. We can see that both *-g* and *-b* reduce the proportion of suboptimal node expansion on top of JPS and CJPS (Table 3a), but the improvement is not enough to pay the overhead, so they don't achieve better performance (Table 3b). Thus, in the next experiment, we focus on CJPS.

r (%)	time(s) domain	cjps	jps	speed up
0.0	dao	13.93	13.58	0.97
	bgmaps	4.95	3.49	0.71
	starcraft	37.86	33.12	0.87
	street	17.14	13.44	0.78
	iron	167.35	175.17	1.05
0.1	dao	15.52	17.22	1.11
	bgmaps	6.49	7.51	1.16
	starcraft	47.20	68.88	1.46
	street	28.09	98.57	3.51
	iron	341.78	2508.75	7.34

Table 4: Cumulative time per domain on static and dynamic environments.

Exp-3: Domain Maps

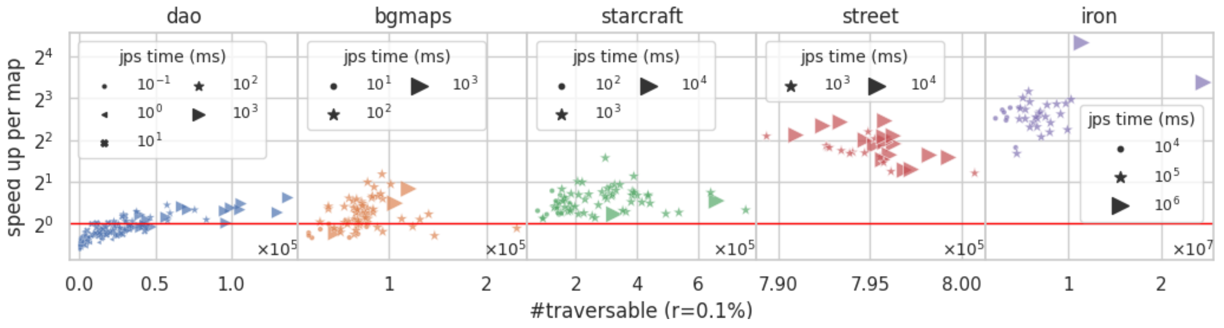
This experiment is to show CJPS performance on public benchmarks from real applications when environments are dynamic. There are three resolutions for city maps (*street*). We pick the highest one (1024) as it is more challenging.

Simulating Dynamic Environments. In real applications, the number of dynamic changes depends on the size of the map and is not very dense. For example, in Starcraft, each player controls at most 200 agents and the number of facilities is usually much smaller than this, while the maps are usually 512×512. Thus, we assume the density of random obstacles is 0.1%. To simulate dynamic environments, we add random obstacles in the same way as for the synthetic maps, assuming the result represents the map after a dynamic change. Removing obstacles has less effect on JPS search unless we analyse the map (e.g. removing a small part of a wall has almost no effect), so we dont use it here.

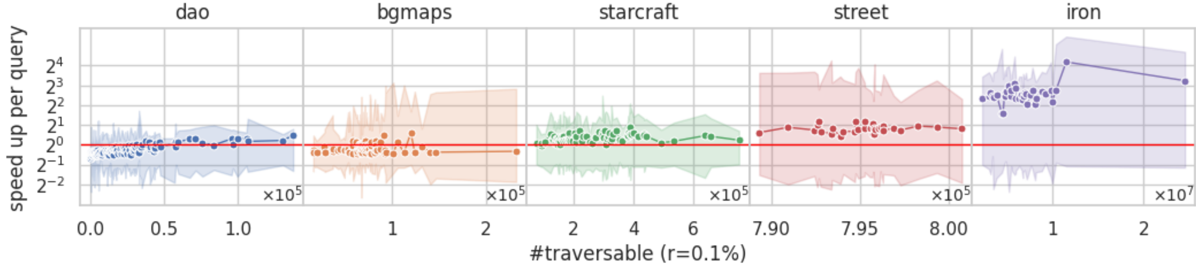
Table 4 shows the total time to finish all queries per domain, we can see that CJPS has no advantage when the environment is static, but wins in dynamic environments, Figure 6 shows detailed results in such environments.

How to Read These Plots. Figure 6a shows speed-up on the cumulative time per map, where markers indicate the order of time to finish all queries. It describes an overview of improvement but misses the distribution of improvement per query. Figure 6b shows a compact distribution of speed-up on queries per map, including min, median and max, but it doesn't directly show what kind of queries are improved. Figure 6c shows the query speed-up in terms of increasing difficulty, i.e., number of JPS node expansion.

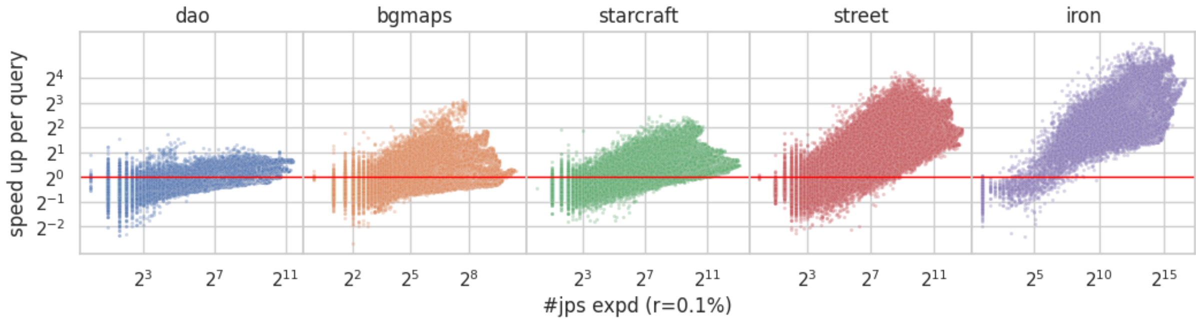
We can see that in *dao* and *bgmaps*, there are no significant improvements. Even though the speed-up of cumulative time can be up to 2 on specific maps, most instances are slower than JPS. The reason is that most maps from these domains are either small or easy. *Dao* has many small dungeon-like maps, and all maps from *bgmaps* are scaled to



(a) Speed-up per map in increasing number of traversable cells.



(b) Compact distribution of speed-up on queries per map. Points represent the median, and the bands represent the min and max values.



(c) Speed-up per query in increasing difficulty (*jps expansion*)

Figure 6: Speed-up on cumulative time ($\frac{\sum Time_{jps}}{\sum Time_{cjps}}$) and queries($\frac{Time_{jps}}{Time_{cjps}}$). The red line is 1, and values above it indicate that CJPS is faster.

512 from a smaller size which have large open spaces and accurate heuristics. In the rest of the domains, as shown in Figure 6b, CJPS has better performance on most maps. In *starcraft*, most queries get improved, i.e., the median point is above red line. In *street*, the median speed-up factors are about 2. In *iron*, CJPS is more than 4x faster on most maps, and can be up to 16x on one large map. According to Figure 6c, an encouraging feature is that, when it is improving CJPS becomes more effective, the harder the query is.

Conclusion and Future Work

In this paper, we study the pathological behaviours of *JPS* and propose a new approach, *CJPS*, which effectively resolves these issues and convincingly improves JPS performance in dynamic environments. Although JPS still wins when the environment is static, offline-based methods should be applied in this case.

An interesting future direction is grid-based pathfinding with higher dimension, e.g. 3D voxel grid map (Nobes et al. 2022) or 2D grid maps with temporal obstacles (Hu et al. 2021), in the latter, obstacles can move and environments are changing during the query. These problems have a larger search space and more symmetries, thus there are more opportunities for CJPS to improve over the baseline algorithm.

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.

Delling, D.; Goldberg, A. V.; Pajor, T.; and Werneck, R. F. 2017. Customizable Route Planning in Road Networks. *Transportation Science*, 51(2): 566–591.

Goldberg, A. V.; and Harrelson, C. 2005. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, 156–165.

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.

Harabor, D.; Hechenberger, R.; and Jahn, T. 2022. Benchmarks for Pathfinding Search: Iron Harvest. In Chrpa, L.; and Saetti, A., eds., *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, 218–222. AAAI Press.

Harabor, D. D.; and Grastien, A. 2014. Improving Jump Point Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 128–135.

Hechenberger, R.; Stuckey, P. J.; Harabor, D.; Le Bodic, P.; and Cheema, M. A. 2020. Online computation of euclidean shortest paths in two dimensions. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 134–142.

Hu, S.; Harabor, D. D.; Gange, G.; Stuckey, P. J.; and Sturtevant, N. R. 2021. Jump Point Search with Temporal Obstacles. In Biundo, S.; Do, M.; Goldman, R.; Katz, M.; Yang, Q.; and Zhuo, H. H., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, 184–191. AAAI Press.

Mahéo, A.; Zhao, S.; Afzaal, H.; Harabor, D.; Stuckey, P. J.; and Wallace, M. 2021. Customised Shortest Paths Using a Distributed Reverse Oracle. In Ma, H.; and Serina, I., eds., *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, 79–87. AAAI Press.

Nobes, T. K.; Harabor, D.; Wybrow, M.; and Walsh, S. D. C. 2022. The JPS Pathfinding System in 3D. In Chrpa, L.; and Saetti, A., eds., *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, 145–152. AAAI Press.

Sturtevant, N. 2012. Benchmarks for Grid-Based Pathfinding. *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 Boutilier, C., ed., *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 609–614.

Sturtevant, N. R.; and Rabin, S. 2016. Canonical Orderings on Grids. In Kambhampati, S., ed., *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 683–689. IJCAI/AAAI Press.

Sturtevant, N. R.; Traish, J. M.; Tulip, J. R.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The Grid-Based Path Planning Competition: 2014 Entries and Results. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search (SOCS-15)*, 241–251.

Traish, J. M.; Tulip, J. R.; and Moore, W. 2016. Optimization Using Boundary Lookup Jump Point Search. *IEEE Trans. Comput. Intell. AI Games*, 8(3): 268–277.