Regarding Jump Point Search and Subgoal Graphs

Daniel D. Harabor[1] , **Tansel Uras**[2] , **Peter J. Stuckey**[1] , **Sven Koenig**[2]

[1]Faculty of Information Technology, Monash University, Melbourne, Australia
{daniel.harabor, peter.stuckey}@monash.edu

[2]Computer Science Department, University of Southern California, Los Angeles, USA
{turas, skoenig}@usc.edu

Abstract

In this paper, we define Jump Point Graphs (JP), a preprocessing-based path-planning technique similar to Subgoal Graphs (SG). JP allows for the first time the combination of Jump Point Search style pruning in the context of abstraction-based speedup techniques, such as Contraction Hierarchies. We compare JP with SG and its variants and report new state-of-the-art results for grid-based pathfinding.

Introduction

Subgoal Graphs (SG) [Uras *et al.*, 2013] and Jump Point Search (JPS) [Harabor and Grastien, 2011] are two popular and successful *symmetry-breaking* algorithms for optimal pathfinding on grid maps. Similar in spirit, these methods proceed quite differently in practice, and the relationship between them, in terms of strengths and weaknesses, has not been well understood. We address this gap in the literature with a new graph-theoretic description of JPS and a direct comparative analysis with SGs.

Our first result shows that JPS, like SG, connects pairs of vertices to one another using a *direct reachability relation*. Moreover, when pairs of vertices are not direct reachable, JPS guarantees that the path is *covered* by a distinguished set of vertices, namely, the set of all jump points. One disadvantage of JPS, compared to SG, is its dependence on grid-scanning and target-detection operations. JPS performs these operations repeatedly during vertex expansions, and their necessity complicates the integration of JPS with in-principle orthogonal speedup techniques.

Our second result is to overcome these limitations by extracting a graph of all jump points. The new algorithm — Jump Point Graphs (JP) — has the same strong symmetry-breaking properties as Jump Point Search but only refers to the input grid while inserting the start and target.

Our third result is an empirical evaluation that shows JP can be fruitfully combined with a variety of orthogonal speedup techniques, including bi-directional search and Contraction Hierarchies [Geisberger *et al.*, 2008]. We also compare JP and a number of variants against leading methods from the literature, including SG. We find that JP can significantly reduce the average number of vertex expansions and also average runtime on some popular grid benchmarks.

Preliminaries and Notation

We consider shortest path search on an 8-neighbour grid graph $G = (V, E)$ constructed from a grid map $\mathbf{M}$. Each cell $n \in \mathbf{M}$ is marked as blocked or unblocked. There are 8 available *move* actions $\vec{v}$ (equiv. *directions*), which serve to transition the search from one cell to the next. Straight moves cost 1 and diagonal moves $\sqrt{2}$. A move $\vec{v}$ from a cell n to n' is unblocked (that is, $(n, n') \in E$) iff n and n' are adjacent unblocked cells and, if $\vec{v}$ is diagonal, then there is no blocked cell adjacent to both n and n' (*no-corner-cutting* constraint).

We say that a diagonal direction $\vec{d}$ results from combining two cardinal directions; i.e. $\vec{d} = \vec{c_1} + \vec{c_2}$. Directions $\vec{v}_1$ and $\vec{v}_2$ are *perpendicular*, written $\vec{v}_1 \perp \vec{v}_2$, when the angle between them is a right angle. We also use the algebra $n' = n + k_1 \times \vec{v_1} + k_2 \times \vec{v_2}$, with k_1 and k_2 as integers, to say that cell n' is reached from cell n with k_1 moves in direction $\vec{v_1}$ followed by k_2 moves in direction $\vec{v_2}$. We sometimes refer to cardinal directions by name: *U(p), D(own), L(eft)*, and *R(ight)*.

A *path* $\pi = \langle n_1 = s, \ldots, n_k = t \rangle$ is an ordered sequence of cells from a distinguished cell called the *start* = s to a distinguished cell called the *target* = t such that there exists a valid move to connect every pair of adjacent cells on the path. When discussing paths, we sometimes refer to their valid *continuations*. A continuation is a move $\vec{v}$ whose application extends the path by one or more cells. For example, $\pi + k \times \vec{v}$ extends path π with k moves in direction $\vec{v}$.

We refer to the (8-neighbor) grid graph constructed from $\mathbf{M}$ by assuming that all cells are unblocked as *freespace*. We refer to a shortest path π in freespace as a *freespace-shortest* path, and say that π is unblocked (on G) iff it is also a path on G. We use a heuristic function h to measure distances in freespace, also referred to as the *Octile distance heuristic*.

Subgoal Graphs

Subgoal Graphs [Uras *et al.*, 2013; Uras and Koenig, 2017] (SGs) is the name of an abstraction-based algorithm developed for fast and optimal pathfinding in directed weighted graphs. During an offline preprocessing step, a distinguished set of vertices, known as *subgoals*, are identified. Each subgoal appears as a vertex in an eponymous SG, and each edge in an SG corresponds to a shortest graph path. However, edges are only added to the SG between pairs of subgoals that satisfy a given *reachability relation*.

We now adapt definitions from [Uras and Koenig, 2017]:

Definition 1. A reachability relation on $G = (V, E)$ is a relation $R \subseteq V \times V$ that satisfies:

1. $\forall s \in V, (s, s) \in R$
2. $\forall (s, t) \in E, (s, t) \in R$ (edge property)

Definition 2. $S \subseteq V$ is an R-shortest-path-cover (R-SPC) on G iff, for all $s, t \in V$, if $(s, t) \notin R$, then at least one shortest s-t path on G passes through some $v \in S$.

Definition 3. t is Direct-R-reachable (DR) from s (with respect to S) on G iff $(s, t) \in R$ and no shortest s-t path on G passes through any $v \in S$.

Definition 4. $G_S = (S, E_S)$ is a subgoal graph on G with respect to R iff S is an R-SPC on G and, $\forall u \neq v \in V$, $(u, v) \in E_S$ iff v is direct-R-reachable from u (with respect to S) on G.

These definitions are adapted from [Uras *et al.*, 2013]:

Definition 5. An unblocked cell s is a *convex corner* (of a blocked cell) iff there are two perpendicular cardinal directions $\vec{c}_1$ and $\vec{c}_2$ such that $s + \vec{c}_1 + \vec{c}_2$ is blocked and $s + \vec{c}_1$ and $s + \vec{c}_2$ are not blocked.

Definition 6. Two cells s and t are *freespace-reachable* [**FR**] iff at least one freespace-shortest path between them is unblocked on G. Two cells s and t are *safe-freespace-reachable* [**SFR**] iff all freespace-shortest paths between them are unblocked on G.

SGs are instantiated on grid graphs using SFR as the reachability relation and by selecting S as the set of convex corners of blocked cells. Searching for shortest paths with a SG is a simple three step process sometimes referred to as Connect-Search-Refine:

1. **Connect:** The algorithm temporarily inserts s and t into the SG (assuming they do not already appear) and connects each of them to all DR subgoals and possibly to one another, in case that s and t are themselves DR.
2. **Search:** The algorithm executes a best-first search in the *query* SG from s to t. Figure 3 shows an example.
3. **Refine:** Having found a *subgoal* path such as $\pi = \langle v_0 = s, \ldots, v_k = t\rangle$ the algorithm finds paths between each pair of adjacent subgoals v_i and v_{i+1} on G.

In [Uras and Koenig, 2017], it is shown that the Connect-Search-Refine process using SGs always finds shortest paths between any two vertices. Meanwhile, in [Uras *et al.*, 2013], it is (implicitly) shown that the set of convex corners is an SFR-SPC on G. Using SFR as the reachability relation has a number of benefits:

- Refinement is trivial: For any $(s, t) \in$ SFR, by definition, any freespace-shortest path is also a path on G.
- Edge costs are not stored (they're equal to Octile distance).
- The set of all Direct SFR (**DSFR**) subgoals for any cell can be identified fast using pre-computed distance-to-obstacle data (equiv. clearance values). This data also helps to speed up the Connection phase of the algorithm.

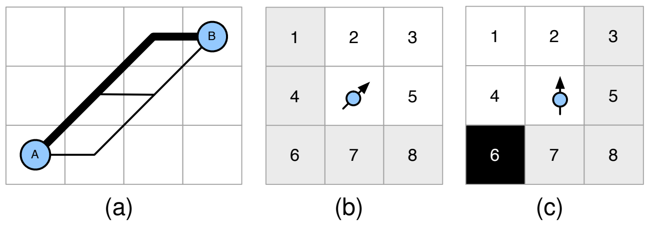

Figure 1: (a) Three equivalent paths connect A and B. JPS *breaks* such symmetries by taking the bold path, where diagonal moves appear first. (b) This *diagonal jump point* has three diagonal-first continuations: 2, 3 and 5. (c) This *straight jump point* also has three diagonal-first continuations: 1, 2, 4. Grey cells are pruned.

Jump Point Search

In uniform-cost grids many *symmetric* paths can connect the same pair of vertices. Figure 1(a) shows an example.

Definition 7. [Harabor and Grastien, 2011] Two grid paths are symmetric iff: **(i)** they share the same start and target and; **(ii)** they have the same cost and; **(iii)** their constituent moves can be re-ordered to derive one path from the other.

In the presence of symmetry, otherwise efficient search algorithms, such as A*, waste time, exploring many equivalent partial paths and making slow progress to the target. The Jump Point Search (JPS) family of algorithms *break* symmetries by maintaining the following invariant during search: when a node n is added or updated on the A* open list, the path to that node is *diagonal-first*. This property means that the only successors generated when n is expanded are those which allow the current path to continue from n in a way that is taut and which guarantees that diagonal moves appear as early as possible. All other successors can be safely pruned.

Definition 8. A path $\pi = \langle s, \ldots, t\rangle$ is *taut* iff every subpath of length two $\langle n_{i-1}, n_i, n_{i+1}\rangle \in \pi$ is also a shortest path.

Definition 9. [Harabor and Grastien, 2011] A path π is *diagonal-first* iff it contains no straight-to-diagonal turning point $\langle n_{k-1}, n_k, n_{k+1}\rangle$ which could be replaced by a diagonal-to-straight turning point $\langle n_{k-1}, n'_k, n_{k+1}\rangle$ to produce a new valid path on G.

The diagonal-first invariant reduces the branching factor of each expanded node, often to just 0 or 1. Rather than adding such nodes to the open list, JPS immediately processes them as part of a recursive and eponymous *jumping* procedure. Jumping allows the search to generate and expand only a distinguished set of nodes at which diagonal-first paths can branch. These nodes are called *jump points*, of which there exist two different types:

Definition 10. A *straight jump point* is a tuple $(n, \vec{c})$ where n is a cell and $\vec{c}$ is a cardinal travel direction such that, for some $\vec{c'} \perp \vec{c}$: **(i)** $n - \vec{c}$ is unblocked; **(ii)** $n + \vec{c'}$ is unblocked; and **(iii)** $(n - \vec{c}) + \vec{c'}$ is blocked.

Definition 11. A *diagonal jump point* is a tuple $(n, \vec{d})$ where n is a cell and $\vec{d} = \vec{c_1} + \vec{c_2}$ is a diagonal direction such that, for some k: **(i)** $n + k \times \vec{c_1}$ is an unblocked path from n to a cell n' such that $(n', \vec{c_1})$ is a straight jump point or; **(ii)** $n + k \times \vec{c_2}$

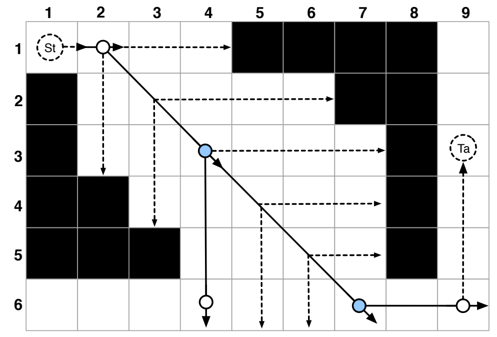

Figure 2: JPS searches from St to Ta. Diagonal jump points are filled, straight jump points are not. Dashed lines indicate recursive row and column scans. Here St has only one successor: $((2,1), R)$. When expanding this node JPS recurses in three directions: R and D, which produce no successors, and $R+D$ which produces $((4,3), R+D)$. The search continues in this way until $((9,6), R)$, where JPS recurses in direction U and generates the target.

is an unblocked path from n to a cell n' such that $(n', \vec{c_2})$ is a straight jump point; or **(iii)** $n + k \times \vec{c_1}$ or $n + k \times \vec{c_2}$ is an unblocked path from n to the target.

Target Detection: JPS treats the target as a special case. While scanning the grid and jumping from one node to the next, JPS *detects* the target and generates it as a successor.

Figures 1(b) and 1(c) show different jump points and their taut diagonal-first continuations. Figure 2 shows a complete example of pathfinding with JPS. Only diagonal-first paths are explored, and every turning point on each diagonal-first path is a jump point. Together with target detection, these features are sufficient to find a shortest path, from any start to any target, if a path exists at all.

Lemma 1. For every shortest path π on G, there exists an equivalent path π' on G which is diagonal-first. □

Some advantages of JPS compared to SG: (i) it runs entirely online; (ii) it requires no preprocessing or insertion; and (iii) it prunes successors more aggressively and often finds the target faster [Harabor and Grastien, 2014]. Some disadvantages:

- The jumping rules are not easily reversed, which means JPS is implemented only as a uni-directional search;
- JPS constantly refers to the input grid to break symmetries and detect the target. These features make JPS difficult to integrate with speedup techniques such as Contraction Hierarchies [Geisberger *et al.*, 2008].

To overcome these limitations, we are going to develop a new and graph-theoretic perspective of JPS. In the process, we also show that JPS and SG are closely related.

Jump Point Graphs

To begin, we formalise the search space of JPS as a graph G^*.

Definition 12. Let $G = (V, E)$ be a grid graph. Its corresponding *direction-extended grid graph* $G^* = (V^*, E^*)$ is defined as:

- For each vertex $n \in V$ and each possible grid move $\vec{v}$, there exists a vertex $(n, \vec{v}) \in V^*$.
- For each $(n_1, n_2) \in E$ and each pair of grid moves $\vec{v_1}$ and $\vec{v_2}$, there exists $((n_1, \vec{v_1}), (n_2, \vec{v_2})) \in E^*$ iff:
  1. $n_1 + \vec{v_2} = n_2$
  2. $\langle n_1 - \vec{v_1}, n_1, n_2 \rangle$ is diagonal-first and taut.

G^* is analogous to G but each cell appears 8 times: once for each direction. Intuitively, G^* labels each vertex n of G with each possible *incoming* direction $\vec{v}$, and limits the *outgoing* directions $\vec{v}'$ from $(n, \vec{v})$ to be diagonal-first and taut with $\vec{v}$.

Lemma 2. Any path π^* on G^* is diagonal-first and taut.

Proof. G^*'s edges are defined such that (1) for each vertex $(n, \vec{v}) \in V^*$, $(n, \vec{v})$ is reachable only with move $\vec{v}$, and (2) each move corresponding to an outgoing edge of $(n, \vec{v})$ is diagonal-first and taut with v. Thus, for any path π^* on G^*, any two consecutive moves are diagonal-first and taut. □

Lemma 3. Let $\pi = \langle n_0, \ldots, n_k \rangle$ be a diagonal-first and taut path on G. Then, there exists $\vec{v_0}, \ldots, \vec{v_k}$ such that $\pi^* = \langle (n_0, \vec{v_0}), \ldots, (n_k, \vec{v_k}) \rangle$ is a path on G^*.

Proof. The proof follows from Definition 12 by selecting, for $i = 1, \ldots, k$, $\vec{v_i}$ as the direction that satisfies $n_{i-1} + \vec{v_i} = n_i$, and selecting $\vec{v_0} = \vec{v_1}$. □

Recall that, for any $s, t \in V$, there exists a diagonal-first shortest s-t path on G (Lemma 1) which, by Lemma 3, is preserved on G^*. Therefore, we can use G^* for finding shortest s-t paths on G, by treating all $(s, \vec{v})$ as start vertices and treating all $(t, \vec{v})$ as target vertices, for all directions $\vec{v}$.

By definition, G^* contains any (straight or diagonal) jump point on G. We now show that the set of straight jump points is a freespace-reachability-SPC on G^*, by extending the definitions of freespace-shortest paths and freespace-reachability (Definition 6) to G^*. We use the prefix *diagonal-first* to distinguish these definitions on G^* from their counterparts on G, and to better characterise their properties.

Definition 13. The *diagonal-first freespace-shortest path* from a cell n to n' is a freespace-shortest path where all diagonal moves appear before cardinal ones. A vertex $(n', \vec{v'}) \in V^*$ is *diagonal-first freespace reachable* [**DFFR**] from $(n, \vec{v}) \in V^*$ iff a path from $(n, \vec{v})$ to $(n', \vec{v'})$ on G^* corresponds to the diagonal-first freespace-shortest path from n to n'. We use **DDFFR** to denote Direct-DFFR-Reachable.

Theorem 1. Straight jump points form a DFFR-SPC on G^*.

Proof. Let $s^* = (s, \vec{v}), t^* = (t, \vec{v}) \in V^*$, such that an s^*-t^* path exists on G^*. We show that, if $(s^*, t^*) \notin$ DFFR, then a shortest s^*-t^* path on G^* passes through a straight jump point. For contradiction, assume that $(s^*, t^*) \notin$ DFFR, and that no shortest s^*-t^* path on G^* passes through a straight jump point. Let π^* be any shortest s^*-t^* path on G^*. By our assumption, π^* does not pass through a straight jump point. By Lemma 2, π^* is diagonal-first and taut. π^* cannot have a cardinal-to-cardinal turn because, otherwise, it is either non-taut or it passes through a jump point. π^* cannot have a diagonal-to-diagonal turn because, otherwise, it is non-taut. π^* cannot have a cardinal-to-diagonal turn because, otherwise, it is either non-diagonal-first or it passes

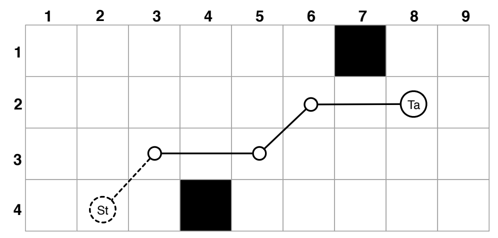

Figure 3: St and Ta are direct reachable with DDFFR but a search with DSFR expands every corner point along the way. Dashed lines are temporary (i.e., inserted) vertices and edges.

through a jump point. Therefore π^* can only have diagonal-to-cardinal turns. There is only one such possible π^*, which corresponds to the diagonal-first freespace-shortest s-t path π. But then, the diagonal-first freespace-shortest s-t path is unblocked, contradicting that $(s^*, t^*) \notin$ DFFR. ☐

We may formalise the relation between SG and JPS as so:

Theorem 2. Subgoal reachability is a more restrictive notion than jump point reachability. Formally, DSFR$\subseteq di($DDFFR$)$, where $di(R)$ is the relation defined as $(s, t) \in di(R)$ iff $\exists \vec{v_1}, \vec{v_2}.\ ((s, \vec{v_1}), (t, \vec{v_2})) \in R$.

Proof. Consider $(s, t) \in$ DSFR. By definition, the diagonal-first freespace-shortest s-t path π is unblocked and does not pass through a subgoal. For some $s^* = (s, \vec{v_1})$ and $t^* = (t, \vec{v_2})$, let π^* be an s^*-t^* path on G^* that corresponds to π. By Lemma 3, such $\vec{v_1}$, $\vec{v_2}$, and π^* exists since π is diagonal-first and taut (freespace-shortest). Since π does not pass through corner points, π^* does not pass through straight jump points. Therefore, $(s, t) \in di($DDFFR$)$. ☐

Theorem 2 suggests that searching over jump points may require fewer expansions per instance than searching over an equivalent SG. Consider that JPS travels from one vertex to the next, stopping only at corner vertices where shortest paths can bend. By comparison, SG search will stop at every corner vertex along the way. Figure 3 shows an example.

Next, consider the subgoal graph $G^*_J = (J, E_J)$ constructed on G^* using the set of straight jump points J as subgoals. From the theory presented in [Uras and Koenig, 2017] and the result of Theorem 1, G^*_J can be used to find shortest paths as long as the following conditions hold:
(1) (Preprocessing) We add edges between every pair of straight jump points that are DDFFR from one another.
(2) We connect the start s: to every straight jump point that is DDFFR from $(s, \vec{v})$, for any direction of $\vec{v}$.
(3) We connect the target t: from every straight jump point for which $(t, \vec{v})$ is DDFFR, for any direction of $\vec{v}$.
(4) We connect the start to the target: iff, for some directions $\vec{v}$, and $\vec{v'}$, $(t, \vec{v'})$ is DDFFR from $(s, \vec{v})$.

We call the resulting method the Jump Point Graph (**JP**). Since JP can be considered a Subgoal Graph on G*, we discuss in the following sections: how to efficiently identify its edges, how to quickly perform the Connection step and how to combine JP with other orthogonal speed-up techniques, including bi-directional search and Contraction Hierarchies.

Algorithm 1: SG connect (black and blue text) and JP forward connect (black and red text).

```
1  function CardinalScan(s, $\vec{c}$, &D)
2      if $\mathbf{C}[s, \vec{c}] > 0$ then
3          add $(s + \vec{c} \times \mathbf{C}[s, \vec{c}], \vec{c})$ to $D$;
4  function DiagonalFirstScan(s, $\vec{d}$, &D)
5      assign $\vec{c_1}, \vec{c_2}$ such that $\vec{c_1} + \vec{c_2} = \vec{d}$;
6      $n \leftarrow s$;
7      while $\mathbf{C}[n, \vec{d}] > 0$ do
8          $n \leftarrow n + \vec{d} \times \mathbf{C}[n, \vec{d}]$;
9          if $n$ is a subgoal then
10             add $n$ to $D$;
11             return;
12         CardinalScan($n$, $\vec{c_1}$, $D$);
13         CardinalScan($n$, $\vec{c_2}$, $D$);
14 function ForwardConnect(s)
15     $D \leftarrow \emptyset$;
16     foreach cardinal grid direction $\vec{c}$ do
17         CardinalScan($s$, $\vec{c}$, $D$);
18     foreach diagonal grid direction $\vec{d}$ do
19         DiagonalFirstScan($s$, $\vec{d}$, $D$);
20     return $D$;
```

Connection Using Clearances

Using precomputed clearances (distances to *important* cells in a certain direction) is an optimisation technique used by both SGs [Uras *et al.*, 2013] and JPS [Harabor and Grastien, 2012; 2014] to improve query times. For SG, clearances help to speed-up the process of identifying a set of DSFR subgoals from any given cell, and they are used when constructing the graph and when connecting the start and target to it. For JPS, clearances are used for quickly scanning the grid to generate jump-point successors. In this section, we show that both algorithms use clearances in a very similar way (the similarities and differences are shown in Algorithm 1). We also show that SGs can be constructed, using clearances, in an amount of time (and using an amount of space) that is linear with respect to the of the size of the input grid.

Algorithm 1 can be considered as a systematic scan of all diagonal-first freespace trajectories that originate from a given source cell s: $\vec{c}$ cardinal scans (lines 1-3) scan those trajectories where the first move is $\vec{c}$, by repeatedly moving in direction $\vec{c}$. The $\vec{d} = \vec{c_1} + \vec{c_2}$ diagonal-first scans (lines 4-13) scan those trajectories where the first move is diagonal, by repeatedly moving in direction $\vec{d}$ and branching off into $\vec{c_1}$ and $\vec{c_2}$ cardinal scans at every visited cell. It uses precomputed *clearance* values in both cardinal and diagonal directions to speed-up the scans. A $\vec{c}$ cardinal clearance $\mathbf{C}[n, \vec{c}]$ from a cell n is the number of $\vec{c}$ cardinal moves that can be made from n to reach an *important* cell m (subgoals for SG connection, jump points with direction $\vec{c}$ for JP connection). If such m does not exist, $\mathbf{C}[n, \vec{c}]$ is 0. Using cardinal clearances reduces cardinal scans to a simple table look-up (line 3) that requires O(1) time, and the scanning algorithm reduces to a sequence of clearance look-ups along the diagonals that extend from s. A $\vec{d}$ diagonal clearance $\mathbf{C}[n, \vec{d}]$ from a cell n is the number

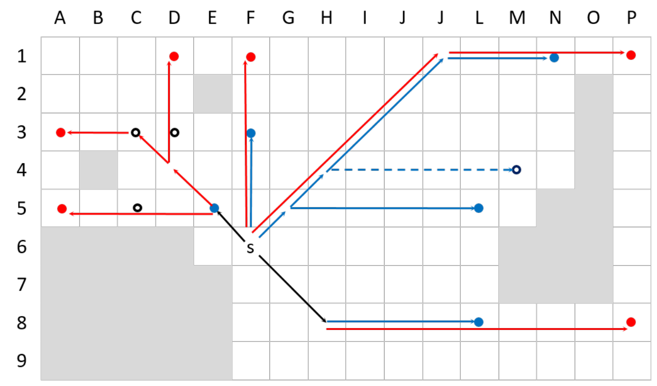

Figure 4: SG and JP connecting the start node s. Blue discs indicate cells that are DSFR from s while red discs indicate straight jump points DDFFR from s. Black circles: convex corners containing only non-DSFR subgoal or non-DDFFR straight jump points. Blue lines indicate clearance look-ups by SG connect while red lines indicate clearance look-ups by JP connect. Black lines: clearance look-ups by both.

of $\vec{d}$ diagonal moves that can be made from n to reach a cell m where $\mathbf{C}[m, \vec{c_1}] > 0$ or $\mathbf{C}[m, \vec{c_2}] > 0$ (that is, the row or column of m contains an *important* cell). $\mathbf{C}[n, \vec{d}] = 0$ if such m does not exist. Diagonal clearances allow one to quickly traverse the diagonal by skipping the rows and columns that do not contain *important* cells.

The main difference between SG connect and JP forward connect is when they terminate their scans. Namely, a $\vec{c}$ cardinal or $\vec{d}$ diagonal-first scan of SG connect stops immediately when it finds a subgoal (as any continuation of the scan can only find subgoals that are not DSFR), whereas a $\vec{c}$ cardinal scan of JP stops when it finds a jump point with direction $\vec{c}$. (The $\vec{d}$ diagonal-first scan can only stop when it finds a jump point with direction $\vec{d}$, but such jump points are not straight-jump points and are therefore not included in JP). This difference can be observed in the example in Figure 4, and is reflected in lines 9-11 of Algorithm 1 (and partially obscured since it is encoded in the cardinal, and, implicitly, diagonal clearances). Another difference, which is omitted from Algorithm 1, is a small optimisation that allows SG connect to prune any non-DSFR subgoals from the scan , which allows constructing the edges of SG exactly (an example is shown as a dashed blue line in Figure 4). This difference is not relevant to our discussion, and we refer the interested reader to [Uras *et al.*, 2013] for further details.

Theorem 3. Using cardinal clearances, an explicit SG can be constructed on a W $\times$ H grid in time O(WH), and therefore its number of edges is O(WH). ☐

Proof. The set of subgoals can be identified in time O(WH) since it suffices to check the eight surrounding cells to determine whether a cell is a convex corner. Cardinal clearances can be computed in time O(WH) by scanning the grid row by row or column by column while maintaining the clearance to the last encountered convex corner. The edges of a SG can be identified in O(WH) time, by running Algorithm 1 from every subgoal, as we discuss below.

Suppose that we perform a $\vec{c}$ cardinal scan from every subgoal. Since each cardinal scan requires O(1) time to look-up a clearance value, and since there are O(WH) subgoals, the total time required for all $\vec{c}$ cardinal scans is O(WH). Suppose that we perform a diagonal-first $\vec{d} = \vec{c_1} + \vec{c_2}$ scan from every subgoal. Each scan moves $\vec{d}$ diagonally from its source s and, for each visited cell, performs in O(1) time a $\vec{c_1}$ and $\vec{c_2}$ cardinal clearance look-up The $\vec{d}$ diagonal that extends from each subgoal stops when another subgoal is found. Therefore, any grid cell is visited at most once by moving $\vec{d}$ diagonally from all the subgoals during their $\vec{d}$ diagonal-first scans. Since each cell is visited at most once, each visited cell requires O(1) time, and there are O(WH) cells, $\vec{d}$ diagonal-first scans from every subgoal requires O(WH) time. Since there are a constant number of directions, the total effort for performing all the scans from every subgoal takes O(WH) time. ☐

This result does not apply to constructing JP, however, as JP diagonal-first scans do not necessarily terminate at convex corners and, as a result, some cells can be visited multiple times by JP $\vec{d}$ diagonal-first scans from different cells. Figure 4 shows an example: The cells F6 and E5 both contain a jump point in the Up direction. The Up+Left diagonal-first scan from (F6, Up) does not stop at E5 since E5 does not have a jump point (F5, Up+Left).

Diagonal Merged Jump Point Graphs

In this section we describe a variant of JP that achieves O(WH) preprocessing time and contains O(WH) edges, similar to SG. As hinted in the previous section and in the proof of Theorem 3, this new variant should terminate its $\vec{d}$ diagonal-first scans when the scan reaches a convex corner. This can be achieved by adding additional jump points in diagonal directions at every convex corner. Namely, for any convex corner n where $n - \vec{c_1}$ and $n - \vec{c_2}$ is unblocked but $n - \vec{d}$ is blocked (with $\vec{d} = \vec{c_1} + \vec{c_2}$ diagonal), we add the jump points $(n, \vec{c_1} - \vec{c_2})$ and $(n, \vec{c_2} - \vec{c_1})$. This variant still contains all the straight jump-points and, therefore, its set of jump points is a DFFR-SPC. The only differences is that, by adding more jump points, we allow Algorithm 1 to terminate its scans earlier. We call this variant JPD. For instance, in Figure 4, adding the diagonal jump point (E5, Up+Left) results in the Up+Left diagonal-first scan from (F6, Up) to terminate at E5, the first convex corner that it scans.

At first glance JPD seems to add many nodes to the jump point graph. However, each new diagonal jump point is equivalent to an existing straight jump point. In Figure 4 we can see that (E5,Up+Left) has the same successors as (E5,Up). Hence we merge these two jump points in JPD into a single jump point without changing the graph connectivity. Since each new diagonal jump point is merged with an existing straight jump point, JPD has no more jump points than JP. Notice however that JPD search may stop at corner points where JP search would not. We will see this in experiments.

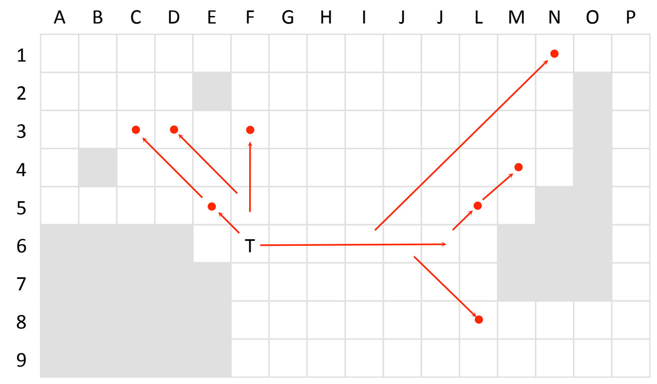

Figure 5: To insert the target JP performs cardinal-first scans to identify all incoming jump points. Unlike start insertion, diagonal recursions of JP (but not JPD) target insertion do not terminate at the first (incoming) jump point.

Theorem 4. Using cardinal clearances, JPD can be constructed using an amount of time and space that is O(WH).

Proof. The proof is similar to that of Theorem 3. □

Searching with Jump Point Graphs

JPs and JPDs use a similar Connect-Search-Refine procedure as SGs. We discuss the main steps:

Start Insertion: To begin, we look for jump point successors of s in each of the 4 cardinal directions. The cost is four constant-time calls to **C**, the set of clearance values stored for every grid cell. We then jump (equiv. recurse) from s in each of the 4 diagonal directions. At each step we look for straight jump point successors at a cost of 2 further constant-time lookups into **C**. For both JP and JPD, the total complexity is linear in the smaller of the map dimensions H and W.

Target Insertion: This operation is similar to start insertion but in reverse: we identify a set of predecessors which covers every *incoming* diagonal-first shortest path, from all other nodes to t. Computing these successors fast requires a set of *reverse clearance values*. A reverse clearance $\mho[(x, y), \vec{v}]$ measures the distance *from* a jump point (or from a blocked cell, if no jump point exists) to cell (x, y) in direction $-\vec{v}$. In JP, the total complexity of target insertion with reverse-clearances is worst-case linear in the size of the grid; i.e., we visit every vertex in the row and column of the target and we recurse along every intersecting diagonal. In JPD, the total complexity is linear in the smaller of the map dimensions H and W, similar to start insertion, since we do not recurse when scanning diagonals. Figure 5 shows JP target insertion. Both JP and JPD scan E5, but while JP recurses to C3, the insertion of JPD does not.

Direct Reachable Paths: Sometimes the shortest path between the start and the target may not involve any vertex from the DFFR-SPC. These cases can be handled similar to SG: by applying target detection during insertion of the start node.

Search: Once the start and target are inserted, we run an optimal search (e.g. A* or bi-directional variant) in the JP or JPD, from the start to the target. The resulting path π is made up entirely of jump points. A main advantage of this representation is that there is no need for explicit refinement: for every pair of adjacent vertices $n_i, n_{i+1} \in \pi$, there always exists a diagonal-first sequence of unblocked cells in the grid, which can be generated without search.

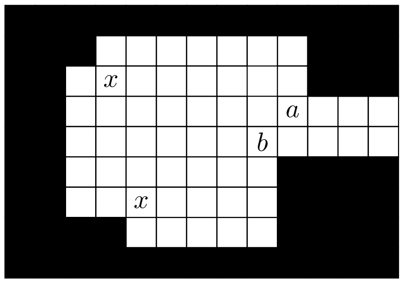

Figure 6: A concave graph illustrating convex corner points.

Avoidance Tables: We now discuss how to speed up insertion of the start and target. Notice in Figure 6 that there are 4 convex corner points, and 8 jump points, two at each marked location. Yet only the jump points at a and b are interesting: the x jump points cannot be reached except if we connect to them during the insertion of the start or target.

In graphs with concave components, there can be a large number of these uninteresting jump points. To reduce the overhead of start and target insertion, we introduce an *avoidance table* for such jump points. When we insert the start, we mark in the avoidance table each such jump point that it connects to. When inserting the target, if we try to connect to a jump point in the avoidance table which is not marked, we omit the connection (since the jump point has no predecessors including the start).

Experimental Setup

We run experiments on a large set of standard grid-based pathfinding benchmarks [Sturtevant, 2012] and on the subset of instances from the 2014 Grid-based Path Planning Competition (GPPC). Our reference point in this comparison is **CH-G**, a fast implementation of Contraction Hierarchies [Geisberger *et al.*, 2008] which appears undominated at the competition. This algorithm stores no downward edges and uses bi-directional Dijkstra search to find a path. It also benefits from stall-on-demand, a common optimisation. To refine a path CH-G uses midpointer unpacking.

Our main experiment compares the following algorithms:

- **SG**, also called Subgoal Simple in [Uras *et al.*, 2013];
- **JP** and **JPD**, as described in the current paper;
- **CH-SG**, **CH-JP** and **CH-JPD**, all of which apply CH on top of each basic method.

We run all algorithms on a 3.6GHz Intel Core i7-7700 CPU with 32GB of RAM. Our implementations are based on C++ code from CH-SG [Uras and Koenig, 2018]. From this common base we derive all algorithms except **CH-G**. Similar to [Uras and Koenig, 2018], all searches use the Octile distance heuristic and a binary heap priority queue. Unlike [Uras and Koenig, 2018], SG and CH-SG store clearances in all eight directions per grid cell, as opposed to only four. JP and

CH-JP store clearances in both the forward and backward directions, for a total of 16 clearances per cell. Each clearance is stored using only 1 byte and clearance values longer than 255 are extracted in multiple steps.

Results

We begin with Table 1 which compares the different methods on the set of maps and instances from GPPC 2014 that are drawn from games. Games are one of the principal uses of grid maps and here we clearly see the impact of Theorem 2: JP has many fewer edges than SG, it expands many fewer nodes, it has a smaller branching factor and its runtime performance is faster by several factors. In combination with Contraction Hierarchies the performance gap is reduced. Here we make three observations: (i) most of the advantage that CH-JP has vs. CH-SG comes from having a faster Search and faster Refine stage. (ii) the Connect stage of both algorithms is comparable; i.e. despite JP target insertion having worst-case linear time requirements we find that the procedure is fast in practice; (iii) both CH-JP and CH-SG faster than CH-G, a method that is undominated at the competition.

In Table 2 we include additional benchmarks from Sturtevant's well known repository and we consider the full set of maps and instances from GPPC 2014. We again observe that JP and JPD achieve shorter query times than SG. However, when combined with CH, we observe that CH-SG is faster than CH-JP and CH-JPD on each of: random, room, and maze maps with small corridor widths. We think that JP and JPD do not combine with CH as well as SG for the following reason: In JP and JPD, multiple jump points may share a cell, but are treated as different vertices. Moreover, the shortest paths between some pairs of jump points in JP or JPD might be longer than the shortest path between the cells that contain these jump points (for instance, if the start jump point has the wrong direction to lead the search to the target jump point). When applying CH to JP or JPD without prior knowledge of this fact, the CH construction algorithm tries to preserve these suboptimal paths, which can introduce redundant edges. Our CH-JP and CH-JPD implementations eliminate most of these redundant edges during contractions, by performing distance queries on JP or JPD to check whether new shortcuts are redundant. We consider other optimisations to improve the combination of JP or JPD with CH as a future work.

vs. JPS+: JPS+ is a preprocessing-based variant of online JPS which stores clearance value data to speed up search. Although not a direct competitor here we may compare against this algorithm based on its performance at GPPC 2014. At the competition JPS+ is reported to be 21.36x slower than the same implementation of of CH-G appearing in our experiments [Sturtevant *et al.*, 2015]. In Table 2, row `GPPC-all` we see that JP is slower than CH-G by a factor of 7.48. Observe also that JP is *faster* than CH-G for all subsets of benchmarks at GPPC except `random`. Here JP is two orders slower than CH-G, which accounts for the overall result.

vs. JPS+BB: JPS+BB [Rabin and Sturtevant, 2016] is a combination of jump point search and bounding boxes. Running the GPPC benchmarks the authors found JPS+BB to be 2.42 faster than CH-G and required 16% less storage, but 3.14 longer preprocessing time. In comparison CH-JP is 4.83 times faster than CH-G, uses 49% less storage, and has 51% shorter preprocessing. Hence CH-JP improves upon JPS+BB.

Conclusion

In summary, we introduce JP which is uniformly better than SG, even on random maps where there end up being many more jump points than subgoals. We allow the use of contraction hierarchies for jump point search for the first time, improving the state-of-the-art in path planning for the important category of game maps.

Acknowledgements

Daniel Harabor is funded by the Australian Research Council grant DE160100007. Research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1724392, 1409987, 1817189 and 1837779. Views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organisations, agencies or the U.S. government.

References

[Geisberger *et al.*, 2008] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333. Springer, 2008.

[Harabor and Grastien, 2011] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *AAAI*, 2011.

[Harabor and Grastien, 2012] Daniel Harabor and Alban Grastien. The JPS+ Pathfinding System. In *SoCS*, 2012.

[Harabor and Grastien, 2014] Daniel Harabor and Alban Grastien. Improving jump point search. In *ICAPS*, pages 128–135, 2014.

[Rabin and Sturtevant, 2016] Steve Rabin and Nathan Sturtevant. Combining bounding boxes and jps to prune grid pathfinding. In *AAAI*, 2016.

[Sturtevant *et al.*, 2015] Nathan R Sturtevant, Jason Traish, James Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. The grid-based path planning competition: 2014 entries and results. In *SoCS*, 2015.

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

[Uras and Koenig, 2017] Tansel Uras and Sven Koenig. Feasibility study: Subgoal graphs on state lattices. In *SoCS*, pages 100–108, 2017.

[Uras and Koenig, 2018] Tansel Uras and Sven Koenig. Understanding subgoal graphs by augmenting contraction hierarchies. In *IJCAI*, pages 1506–1513, 2018.

[Uras *et al.*, 2013] Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *ICAPS*, 2013.

	Prep. time (s)	Memory (MB)	$\lvert V\rvert$	$\lvert E\rvert$	Expanded	Successors per exp.	Time (μs)			
							Connect	Search	Refine	Total
A*	-	2.81	48250	367992	21840.03	7.61	-	4085.79	-	4085.79
CH-G	37.12	4.43	48250	387136	139.15	7.73	-	55.45	23.31	78.76
SG	0.01	1.68	1364	20196	703.34	14.41	5.37	119.02	2.64	127.03
JP	0.02	3.24	2764	8799	186.16	1.78	5.40	22.46	1.64	29.50
JPD	0.02	3.23	2753	7021	230.12	1.44	4.82	24.55	1.93	31.30
CH-SG	11.94	1.78	1364	11426	59.12	9.01	4.99	13.12	4.85	22.96
CH-JP	0.57	3.35	2764	9742	34.72	2.13	5.34	5.42	3.33	14.09
CH-JPD	0.23	3.33	2753	8367	36.98	1.93	4.80	5.54	3.81	14.14

Table 1: Comparison on the subset of GPPC 2014 maps and instances which are drawn from computer games. There are 27 maps from DAO, 67 maps from DA2 and 11 from StarCraft. We give average graph size for each algorithm (A* runs on input grid maps) and we report performance metrics for nodes (average expansions, average successors per expansion) and runtime (total as well as for each of Connect-Search-Refine). Columns Prep and Memory indicate overheads.

	Edges relative to G (%)							Speed up over A* on G						
	CH-G	SG	JP	JPD	CH-SG	CH-JP	CH-JPD	CH-G	SG	JP	JPD	CH-SG	CH-JP	CH-JPD
bg	89.2	13.4	5.8	**4.1**	8.1	6.6	5.8	21.2	10.1	41.9	34.2	34.7	**49.9**	49.6
bg-512	111.4	1.4	0.4	**0.4**	0.8	0.5	0.5	17.6	59.1	220.5	207.6	161.4	246.3	**246.8**
dao	98.6	6.3	3.2	**2.5**	3.8	3.9	3.4	40.9	24.6	80.3	77.1	121.4	159.2	**161.8**
da2	93.1	4.8	2.1	**1.9**	2.8	2.4	2.3	30.0	25.3	92.6	83.2	74.1	**105.6**	104.5
sc	110.9	5.1	2.4	**1.9**	2.8	2.6	2.1	60.4	41.5	210.1	199.0	289.6	498.4	**506.8**
wc3-512	108.7	1.6	0.9	**0.7**	0.9	1.0	0.9	13.5	82.2	212.7	212.5	190.4	230.1	**237.4**
maze-1	85.8	27.6	38.4	38.4	**23.5**	60.4	60.4	91.1	4.0	4.1	4.2	**159.6**	129.8	128.7
maze-2	62.1	6.6	5.9	5.9	**5.0**	8.7	8.7	164.1	11.4	16.4	16.4	**376.6**	351.8	354.8
maze-4	65.6	1.8	1.4	1.4	**1.2**	1.8	1.9	222.6	43.6	102.7	91.2	772.3	**864.6**	826.4
maze-8	80.5	0.4	0.4	0.3	**0.3**	0.4	0.5	225.6	172.6	395.5	359.0	1279.4	**1416.4**	1363.5
maze-16	93.3	0.1	0.1	0.1	**0.1**	0.1	0.1	211.0	598.5	1218.7	1127.0	1917.8	**2114.0**	2030.6
maze-32	101.7	0.0	0.0	0.0	**0.0**	0.0	0.0	158.6	1705.3	2656.6	2531.7	2527.8	**2687.2**	2646.4
random-10	104.7	**30.2**	47.6	34.1	42.6	178.5	135.4	9.2	3.1	3.4	3.6	**35.5**	12.2	16.5
random-15	98.9	**34.8**	50.3	37.6	47.3	174.2	138.5	16.7	2.8	3.0	3.0	**45.6**	16.4	20.0
random-20	92.6	**37.2**	50.9	40.0	47.1	153.9	129.3	28.3	2.6	2.7	2.7	**60.3**	22.8	27.1
random-25	85.5	**38.2**	50.3	41.4	43.6	129.1	113.8	47.7	2.6	2.8	2.7	**84.8**	36.2	40.8
random-30	78.5	**38.3**	49.3	42.4	38.7	105.7	97.3	81.6	2.5	2.9	2.8	**121.7**	59.9	64.9
random-35	71.5	37.7	47.5	42.6	**32.9**	84.0	80.2	154.8	2.6	3.3	3.1	**190.5**	117.8	119.1
random-40	66.6	37.1	46.4	42.8	**28.5**	69.9	68.4	193.8	2.8	4.3	3.8	**204.4**	178.6	167.1
room-8	69.8	**3.2**	4.2	4.1	4.1	10.2	10.1	127.6	23.6	23.7	24.2	**257.3**	152.9	156.1
room-16	81.0	**0.6**	0.8	0.8	0.8	1.8	1.8	158.2	110.8	119.4	119.6	**445.5**	333.0	333.9
room-32	92.9	**0.1**	0.2	0.2	0.2	0.3	0.3	147.1	398.4	449.0	446.6	**761.5**	659.3	659.4
room-64	101.6	0.0	0.0	0.0	**0.0**	0.1	0.1	95.4	1137.4	1208.1	1199.9	**1214.8**	1156.7	1149.2
street-256	103.4	7.4	3.1	**2.1**	4.4	3.8	3.1	7.0	17.8	55.9	52.1	47.0	64.1	**67.1**
street-512	112.5	6.3	1.6	**1.0**	3.5	1.8	1.4	7.8	34.4	151.7	137.5	113.7	198.1	**204.4**
street-1024	118.7	4.8	0.7	**0.4**	2.5	0.7	0.5	9.0	66.8	512.3	418.8	254.2	701.7	**735.9**
GPPC-game	105.2	5.5	2.4	**1.9**	3.1	2.6	2.3	51.9	32.2	138.5	130.5	178.0	**289.9**	288.9
GPPC-maze	88.2	0.3	0.2	0.2	**0.2**	0.3	0.3	480.0	399.5	921.2	821.2	4827.2	**5394.4**	5173.5
GPPC-random	74.8	38.1	48.4	42.7	**35.8**	94.1	88.6	208.7	2.3	2.5	2.4	**328.1**	170.7	176.7
GPPC-room	111.0	0.6	0.9	0.9	**0.5**	1.8	1.7	47.2	261.6	275.9	271.9	**657.2**	408.6	405.2
GPPC-All	98.2	6.0	6.7	5.9	**5.2**	12.6	11.8	305.3	34.8	40.8	39.2	**1757.2**	1477.5	1485.5

Table 2: Number of edges relative to the grid graph G and the speed up over A* on G. All maps from MovingAI benchmarks are included, except for the last 20 of the 30 street-1024 maps, due to high CH-SG preprocessing times.