Robotica: (2020) volume 38, pp. 207–234. © Cambridge University Press 2019
doi:10.1017/S0263574719000560

CWave: Theory and Practice of a Fast Single-source Any-angle Path Planning Algorithm

Dmitry A. Sinyukov†* and Taşkin Padir‡

†*Robotics Engineering Program, Worcester Polytechnic Institute, Worcester, MA 01609, USA*
‡*Electrical and Computer Engineering Department, Northeastern University, Boston, MA 02115, USA*
E-mail: t.padir@northeastern.edu

(Accepted March 27, 2019. First published online: May 15, 2019)

SUMMARY
Path planning on a two-dimensional grid is a well-studied problem in robotics. It usually involves searching for a shortest path between two vertices on a grid given that some of the grid cells are impassable (occupied by obstacles). *Single-source* path planning finds shortest paths from a given source vertex to all other vertices of the grid. Singles-source path planning enhances robot autonomy by calculating multiple possible paths for various navigation scenarios when the destination state is unknown. A high-performance algorithm for single-source any-angle path planning on a grid called CWave is proposed here. *Any-angle* attribute implies that the algorithm calculates paths which can include line segments at any angle, as opposed to standard A* that runs on an 8-connected graph, which permits turns with 45° increments only. The key idea of CWave is to abandon the graph model and operate directly on the grid geometry using discrete geometric primitives (instead of individual vertices) to represent the wave front. In its most basic form (*CWaveInt*), CWave requires only integer arithmetics. *CWaveInt*, however, can accumulate the distance error at turning points. A modified version of CWave (*CWaveFpuSrc*) with minimal usage of floating-point calculations is also developed to eliminate any accumulative errors, which is proven mathematically and experimentally on several maps. The performance of the algorithm on most of the tested maps is demonstrated to be significantly faster than that of Theta*, Lazy Theta*, Field A*, ANYA, Block A*, and A* adapted for single-source planning (on maps with lower number of isolated obstacles, *CWaveFpuSrc* is 2−3 times faster than its fastest tested alternative Block A*). An *N*-threaded implementation (*CWaveN*) of CWave is presented and tested to demonstrate an improved performance (multithreaded implementation is 1.5−3 times faster than single-threaded CWave). The paper discusses foundations and experimental validation of CWave, and presents future work to address the limitations of the current implementations and obtain further performance enhancements.

KEYWORDS: Any-angle path planning; Single-source path planning; High-performance algorithm; Path planning on a grid; Parallelized algorithms.

1. Introduction

Navigation in cluttered and dynamic environments via a low-throughput human–machine interface (LTI; an interface that substantially limits the amount of control information from the operator into the controlled system) remains to be a challenging problem for making robots practical in real-life applications. One example that motivated this work is path planning for a robotic wheelchair when the user intent is inferred via a brain–computer interface or facial gestures.[1,2] In this problem, the user-intended destination is not always deterministically known, but an optimized navigation system

* Corresponding author. E-mail: dsinyukov@wpi.edu

for a given criterion still can be achieved if all possible navigation scenarios are pre-computed. For example, the total time to destination can be minimized. The pre-computation of possible scenarios requires a high-performance algorithm that is able to calculate distances from a given point to all other points on the map. A similar problem arises when the destination is unknown for other reasons. For example, in exploration tasks, each point on a given map may have a certain degree of exploration interest assigned. These values weighed by the distances from the current robot position can determine the robot direction of motion.

The problem of finding the shortest paths from a given vertex to all other vertices on a uniform grid where some grid cells can be blocked is known as the *single-source* shortest path planning problem. This distinguishes it from other three related variations: single-pair, single-destination, and all-pair shortest path problems. They ask, respectively, for the shortest paths between two vertices, all vertices and one destination, and all pairs of vertices on the grid.

In this paper, we present the development of a high-performance algorithm, *CWave*, for single-source any-angle path planning on a two-dimensional (2D) grid. Two-dimensional grid is a popular, but not the only method to represent a 2D environment. Other representations include Navigation Meshes,[3] Circle-Based Waypoint Graphs,[4] and Probabilistic Quadtrees,[5] but these require an additional pre-processing of the map. A 2D grid is simple, easy to use, and is often a direct output of simultaneous localization and mapping (SLAM) algorithms.[6] A grid representation of the environment is also internally used in many autonomous and semi-autonomous robot navigation systems[1,7] and video games.[8]

The common approach to finding the shortest path on a grid is to represent the grid as a 4- or 8-connected graph. Various graph search algorithms can then be employed to find the shortest paths. The problem, however, is that these paths will be suboptimal, because the graph representation of the grid restricts path segment angles to 90° (4-connected graph) or 45° (8-connected graph). Increasing the connectivity of the graph by including second-order neighbors can find paths closer to the optimal, but it cannot solve the fundamental problem arising from the graph representation of the grid. In addition to that, since only 45° turns are allowed, these suboptimal paths are usually unnatural. Path planning algorithms which do not restrict path segment angles are called *any-angle path planning* algorithms. They are used on various mobile robots including such advanced systems as the Mars Rovers Spirit, Opportunity, and Curiosity.[9,10]

Most of the graph search algorithms designed for single-pair path planning can be modified to solve *single-source* path planning problems. Indeed, if the search is not stopped when the destination is reached, it will eventually find distances to all vertices. Thus it makes sense to consider existing any-angle path planning algorithms. It is clear, however, that algorithms that first find a suboptimal path and then smooth it (e.g. A* on a graph with post smoothing[11]) are not suitable for high-performance tasks, because they process each destination separately.

Algorithms which can be easily adapted for single-source path planning include Theta*[12] and its modifications,[13] as well as Field A*.[14] Theta* developers[9] consider these algorithms as interleaving the A* search and the smoothing. These algorithms assume a graph representation of the grid.

Another approach is to first identify special points on the map (usually associated with corners of obstacles), and then do a search in the graph constructed from these points. Visibility graph[15] is a method of this type, but search in this case is typically slow, since the number of edges can grow quadratically in the number of vertices.[16] Another method which can be considered an adaptation of visibility graphs for a grid[17] is Subgoal Graphs.[18] These are shown to be very fast for point-to-point path planning,[17] but they have to pre-process the map, requiring the knowledge of the whole map in advance. Contrarily, CWave has the potential to be applicable for partially known maps evolving as the environment is explored. Initial preprocessing required to construct Subgoal Graphs is reported[19] to be in the order of 100 ms which is about 50 times slower than a single run of CWave on the tested maps. Moreover, Subgoal Graphs, as opposed to CWave, are not optimal in terms of the path length and have not yet been adapted for single-source planning.

Conceptually CWave is a wave-propagation algorithm and, in this sense, is a special case of the Fast Marching Method,[20] where the interface velocity is constant. It is also similar to a well-known wave (Lee) algorithm[21] that deals with octagonal or square waves propagating over 8- or 4-connected graphs, respectively. In case of CWave, however, the wave front has a circular shape (that's what "C" stands for in "CWave") to the extent permitted by the grid. The main idea of CWave is to abandon the graph model and operate directly on the grid geometry using discrete geometric primitives (discrete circular arcs and lines), instead of individual vertices, to represent the wave front.

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

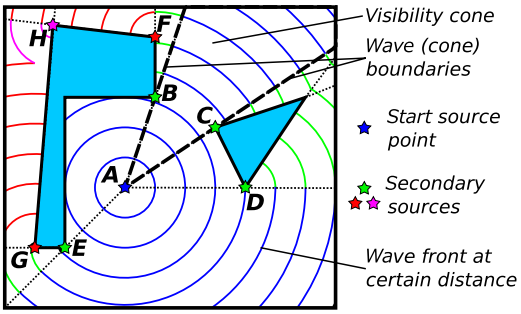

Fig. 1. The gradual expansion of a circular wave from the *start source* point *A* (blue wave) allows to assign distances to all points directly visible from *A*. Then at every point where the wave meets an obstacle (corner points *B*, *C*, *D*, and *E*), a new *source* point can be placed. Simultaneous expansion of circular waves from new *sources* (green waves expanding from points *B*, *C*, *D*, and *E*) allows to further assign distances to points in the bounded area. At a certain moment, some of the waves may merge (e.g. waves expanding from sources *H* and *G*).

Let us first consider a *continuous* space (Fig. 1) where we want to calculate distances from a given point *A* to all other points of the bounded area. The gradual expansion of a circular wave from the *start source* point *A* (blue wave) allows to assign distances to all points directly visible from *A*. Then at every point where the wave meets an obstacle (corner points *B*, *C*, *D*, and *E*), a new *source* point can be placed. Simultaneous expansion of circular waves from new *sources* (green waves expanding from points *B*, *C*, *D*, and *E*) allows to further assign distances to points in the bounded area. At a certain moment, some of the waves may merge (e.g. waves expanding from sources *H* and *G*). This procedure, in the same way as the Lee algorithm, allows to find paths from a given source point to all other reachable points on a 2D map. The circular shape of the wave front, however, in our case ensures that the paths are the shortest. Note that there are two key geometric primitives making this whole construction possible: circular arcs which limit the wave on the front and straight lines which limit the wave on the sides.

This approach in continuous space with polygonal obstacles was first proposed in refs. [22] and [23] as the 'Continuous Dijkstra" paradigm. Mitchell calls each circular wave a "wavelet" and describes three possible events that can occur during a wave front propagation: (1) a wavelet collision with an obstacle, (2) a wavelet disappearance (due to the "closure"), and (3) a wavelet collision with another wavelet.

CWave can be considered an adaptation of the "Continuous Dijkstra" paradigm for a discrete uniform grid. CWave iterates through grid vertices while gradually increasing the distance from a given source vertex. The uniform grid geometry, as a special case of a map with polygonal obstacles, enables specific methods of wave propagation, detection of wavelet collision with obstacles and other wavelets.

We introduce them in the following sections, effectively decomposing the problem. In Section 2, we consider wave propagation on an obstacle-free grid. In Section 3, we introduce simply connected obstacles (wavelet collision with obstacles), and finally Section 4 discusses CWave on a generic map (wavelet overlaps). Performance tests are presented in Section 5. An analysis of the algorithm and future work are discussed in Sections 6 and 7.

When describing the algorithm, we put a certain emphasis on an integer arithmetic implementation *CWaveInt*. While such implementation may not provide any advantage on modern CPUs, especially given that it is subject to a small accumulative distance error (Section 3.6), it enables any-angle path planning on low-cost embedded microcontrollers without floating-point units (FPUs). We then demonstrate how, with minor modifications, the integer-only solution is converted into near-optimal and optimal implementations *CWaveFpuSrc* and *CWaveFpuMerge* (Sections 3.6 and 4.4) that require floating-point operations.

2. CWave on a Grid Without Obstacles

In general, 2D grid is a set of uniformly arranged square *cells*, where each cell can be either occupied or free, and grid *vertices* coincide with the vertices of the square cells (Fig. 2). In this section, we will discuss the most basic case of CWave propagation on a grid without occupied cells. It should be noted that, similar to Theta*, CWave calculates paths between grid vertices not between cell centers.

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

Table I. List of variables used in this paper.

Variable	Description	Type
δ	*Distance error* defined as $\delta = d - r$	Real
ρ	Exact distance of the shortest path from the start source to a given vertex $\rho = s + d$	Real
$\overline{\rho}$	Integer distance value assigned by CWave to a given vertex, approximately equal to 2ρ	Integer
ε	*Error function* used to replace δ to avoid real number calculations	Integer
d	Exact distance from a given vertex (x, y) to its nearest source vertex $d = \sqrt{x^2 + y^2}$	Real
$\overline{d}$	Integer distance value assigned by CWave that is approximately equal to $2d$	Integer
r	Radius of a circle iterated by the midpoint algorithm	Integer
s	Exact distance from the start source to a given source	Real
$\overline{s}$	Rounded $2s$	Integer

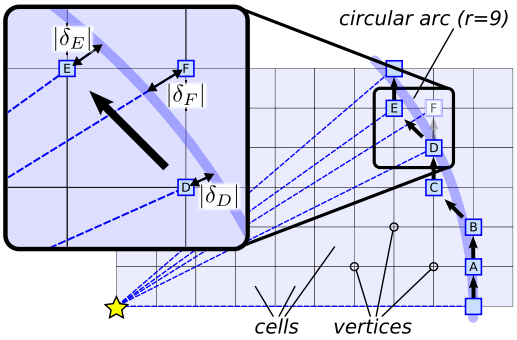

Fig. 2. Iterating through vertices of a circular arc of radius $r = 9$ using MPC algorithm. Zoomed area shows the distance errors.

This is not a completely free "design choice", but it has a strong theoretical motivation. From the calculus of variations,[24] it is known that the shortest path between any two points in a constrained 2D space consists of straight line segments and parts of obstacle boundaries, but the grid model implies that any obstacle is a set of occupied square cells, and thus obstacle boundaries are also straight line segments. Therefore, the shortest path is a polygonal curve with turning points at corners of the occupied cells (at vertices). Placing start and end points of the paths at vertices allows to keep those points in the same set with turning points.

Our goal is to assign each vertex a distance value in such order that it would emulate a propagation of a circular wave. The midpoint circle (MPC) algorithm,[25] also known as *Bresenham's circle algorithm*, can be utilized for this purpose. It was developed to paint out pixels (*cells*) on a digital display in a shape of circular arcs. It is highly efficient, because it requires only integer arithmetics and multiplication by 2 for calculations. In this work, we will employ it to iterate through grid *vertices*, rather than *cells*.

2.1. *Overview of MPC algorithm*

Without loss of generality, we will discuss only octant 1 on XY-plane with the origin at the start vertex. Octant 1 is defined as $x > 0$, $y \geqslant 0$, and $x \geqslant y$. In other octants, the algorithm is similar.

The main idea of MPC algorithm is that in octant 1 when drawing a circular arc of an integer radius r, y-coordinate of each new vertex is always incremented by 1, whereas the x-coordinate is either decremented by 1 (vertices $B \to C$ in Fig. 2) or stays the same (vertices $A \to B$ in Fig. 2). If we define the *distance error* (see Table I for the key variables used in the paper) at any vertex $D(x, y)$ as a difference between the real distance d to D and the circle of integer radius r:

$$\delta_D = \delta_D^r = \delta(x, y, r) = d - r = \sqrt{x^2 + y^2} - r \tag{1}$$

then between two potential candidates (E and F in Fig. 2), vertex E can be chosen over F if

$$|\delta_E| < |\delta_F| \tag{2}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

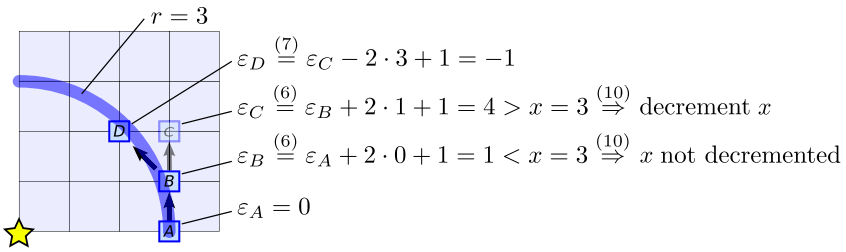

Fig. 3. A simple example that illustrates how MPC algorithm works on a circle of radius 3.

To simplify the calculations, Bresenham introduced another *error function*:

$$\varepsilon_D = \varepsilon_D^r = \varepsilon(x_D, y_D, r) = (x_D^2 + y_D^2) - r^2 = (x^2 + y^2) - r^2 \tag{3}$$

By definition, ε is always an integer, does not require a square root for calculation, and, in the domain of interest, is monotonously related to δ. Thus, condition (2) can be replaced with

$$|\varepsilon_E| < |\varepsilon_F| \tag{4}$$

If we take $D = (x, y)$, then

$$\varepsilon_D = \varepsilon(x, y, r) = x^2 + y^2 - r^2 \tag{5}$$

$$\begin{aligned}\varepsilon_F &= \varepsilon(x, y+1, r) = x^2 + (y+1)^2 - r^2 \\ &= \varepsilon(x, y, r) + 2y + 1 = \varepsilon_D + 2y + 1\end{aligned} \tag{6}$$

$$\begin{aligned}\varepsilon_E &= \varepsilon(x-1, y+1, r) = (x-1)^2 + (y+1)^2 - r^2 \\ &= \varepsilon(x, y+1, r) - 2x + 1 = \varepsilon_F - 2x + 1\end{aligned} \tag{7}$$

To eliminate the absolute value function, we can write (4) as

$$\varepsilon_E^2 - \varepsilon_F^2 < 0 \tag{8}$$

And then, in view of (6) and (7):

$$\begin{aligned}(\varepsilon_F - 2x + 1)^2 - \varepsilon_F^2 &< 0 \\ (\varepsilon_F - 2x + 1 - \varepsilon_F)(\varepsilon_F - 2x + 1 + \varepsilon_F) &< 0 \\ (-2x + 1)(\varepsilon_F - x + 0.5) &< 0\end{aligned} \tag{9}$$

For a positive integer $x > 0$, $(-2x + 1) < 0$, thus the inequality is further simplified:

$$\varepsilon_F > x - 0.5$$

Finally, given that $x = x_D = x_F$ and ε_F are integers, (9) becomes

$$\varepsilon_F \geqslant x_F \tag{10}$$

Condition (10) defines whether x-coordinate needs to be decreased by 1 or not, whereas Eqs. (6) and (7) are used to incrementally calculate ε. Note that only integer addition, subtraction, and bit shifting (multiplication by 2) are used. For circles, the initial value of ε is $\varepsilon_0 = \varepsilon(r, 0, r) = 0$, and for *arcs*, it has to be determined by another method (see Section 3). Figure 3 illustrates how MPC algorithm works on a circle of radius 3.

2.2. *Distance error of Bresenham circles*

From (1), we can express $(x^2 + y^2)$ in terms of δ and r, then by substituting $(x^2 + y^2)$ into (3), we can express *error function* ε as a function of *distance error* δ:

$$\varepsilon(\delta) = \delta^2 + 2r\delta \tag{11}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

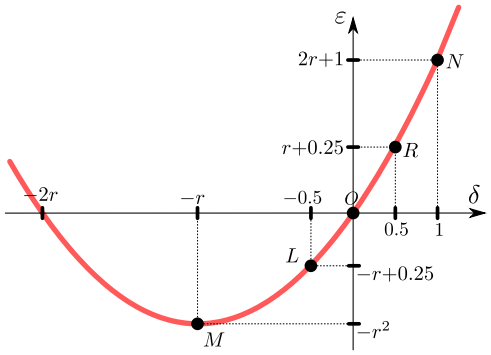

Fig. 4. *Error function* ε as a function of *distance error* δ.

Figure 4 depicts the continuous mapping defined by (11), where point M is the minimum of $\varepsilon(\delta)$, and L, R, and N correspond to δ values of -0.5, 0.5, and 1, respectively. It should be noted that in reality ε can take only integer values, hence δ (the domain of the function) is restricted to the corresponding isolated individual points.

In order to analyze the distance accuracy of the method, it will be necessary to find bounds for the distance error.

Lemma 1. *In all vertices iterated by MPC algorithm,*

$$|\delta| < 0.5 \tag{12}$$

Proof. Given that the initial value of distance error $\delta_0 = \delta(r, 0, r) = 0$, to prove by induction, we need to show that if (12) is true for the distance error at kth step δ_k, then it's also true for δ_{k+1}. From (11), assuming that $|\delta_k| < 0.5$ and taking into account that ε is an integer, we will have

$$-r + 1 \leqslant \varepsilon_k \leqslant r \tag{13}$$

Case 1: Condition (10) is false. Based on the update rule (6),

$$\varepsilon_{k+1} = \varepsilon_k + 2y_k + 1 < x_k \tag{14}$$

This, in combination with (13), implies

$$-r + 2y_k + 2 \leqslant \varepsilon_{k+1} < x_k \tag{15}$$

Now, given that in octant 1, $x_k \leqslant r$ and $y_k \geqslant 0$, (15) implies

$$-r + 0.25 < \varepsilon_{k+1} < r + 0.25 \tag{16}$$

which, based on Fig. 4, entails $|\delta_{k+1}| < 0.5$.

Case 2: Condition (10) is true. Based on the update rules (6) and (7),

$$\begin{aligned} \varepsilon_{k+1} &= \varepsilon_k + 2(y_k - x_k + 1) \\ &= \varepsilon_k + 2(y_{k+1} - x_{k+1} - 1) \geqslant x_k \end{aligned} \tag{17}$$

This, in combination with (13), implies

$$x_k \leqslant \varepsilon_{k+1} < r + 2(y_{k+1} - x_{k+1} - 1) \tag{18}$$

Now, given that in octant 1, $0 \leqslant y_{k+1} \leqslant x_{k+1}$, and thus $(y_{k+1} - x_{k+1} - 1) \leqslant -1$, then

$$0 \leqslant \varepsilon_{k+1} < r - 2 \tag{19}$$

Based on Fig. 4, the last system of inequalities implies that $0 \leqslant \delta_{k+1} < 0.5$.

We just showed that if $|\delta_k| < 0.5$, then $|\delta_{k+1}| < 0.5$.

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

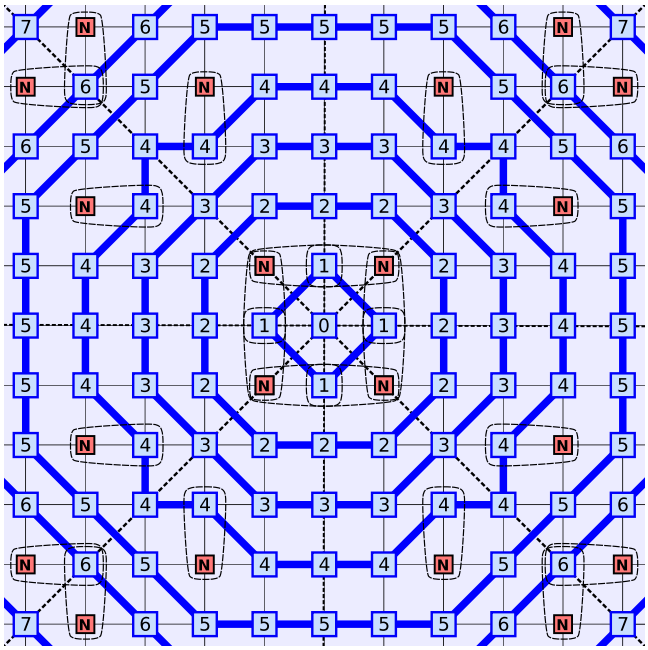

Fig. 5. Circles of integer radii drawn with Bresenham's midpoint algorithm in octant 1. **N** designates NBPs.

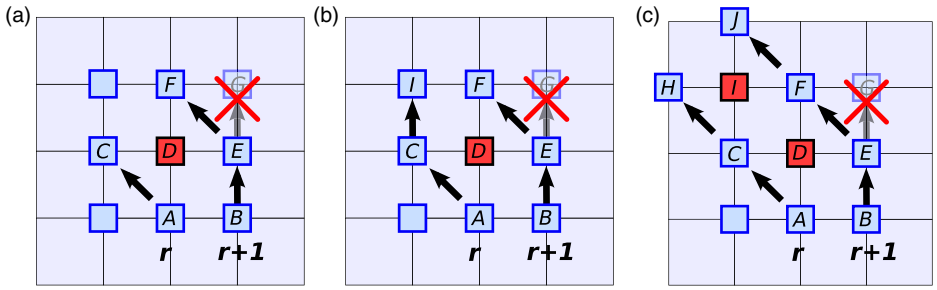

Fig. 6. Formation of NBPs. NBPs are marked with red. (a) Next vertex of $(r+1)$-circle after E is always F; (b) if next vertex after C is I, then no new NBPs; and (c) if next vertex after C is H, then I is a new NBP, and next vertex after F is always J.

2.3. Non-Bresenham points and their properties

It is noted that not all vertices on the grid are accessible by MPC algorithm. Indeed, in Fig. 5, we can see Bresenham circles of integer radii $r \leqslant 7$. Numbers at the vertices designate the radius—vertices that have not been visited are marked with **N**. In this paper, these vertices are referred to as *non-Bresenham points* (NBPs). For path planning, it is important to iterate through all vertices to check their reachability; thus for further analysis, we need to investigate how NBPs are formed.

Let us consider a configuration shown in Fig. 6(a). Here vertices A and B corresponding to circles of radii r and $(r+1)$, respectively, are adjacent, but then on the next iteration r-circle moves diagonally to vertex C, whereas $(r+1)$-circle goes up to vertex E. This leaves vertex $D(x, y)$ unvisited, thus making it an NBP. We will now show that in this configuration, the next vertex for $(r+1)$-circle is always F.

According to condition (10) of MPC algorithm, we just need to prove that $\varepsilon_G^{r+1} \geqslant x$:

$$\varepsilon_G^{r+1} = \varepsilon(x+1, y+1, r+1) = \varepsilon_D^r + 2(x + y - r) + 1 \tag{20}$$

Based on condition (10),

$$\varepsilon_D^r \geqslant x > 0 \tag{21}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

which means that $\delta_D^r > 0$ and $r < \sqrt{x^2 + y^2}$. At the same time, from the triangle inequality, $\sqrt{x^2 + y^2} \leqslant (x + y)$, and thus

$$(x + y - r) > 0 \tag{22}$$

After substituting (22) and (21) into (20), we will have

$$\varepsilon_G^{r+1} > x + 1 > x \tag{23}$$

Now we can explore the genesis of NBPs a bit more. If $\varepsilon_I^r < x_I$ (Fig. 6(b)), then r-circle will proceed to vertex I, and no new NBP will be formed. If, however, $\varepsilon_I^r \geqslant x_I$ (Fig. 6(c)), r-circle will proceed to vertex H, and I will become an NBP. In this case, we can apply the same proof that was used for points D and F, to show that the next vertex of $(r + 1)$-circle will always be J.

2.4. Necessary and sufficient condition for an NBP

Proposition 1. *When condition (10) is checked for some vertex F while iterating through an r-circle, F will be an NBP of the r-circle iff (10) is true and*

$$\varepsilon_F^r \leqslant 2r - x_F \tag{24}$$

Proof. Based on condition (10), in scenario shown in Fig. 6(a), where initial points A and B of r- and $(r+1)$-circles are adjacent, the necessary and sufficient condition for vertex $D(x, y)$ to be an NBP is

$$\begin{cases} \varepsilon_D^r \geqslant x_D \\ \varepsilon_E^{r+1} < x_E \end{cases} \tag{25}$$

Given that $\varepsilon_E^{r+1} = \varepsilon(x + 1, y, r + 1) = \varepsilon_D^r + 2(x - r)$, the second condition in (25) can be written as $\varepsilon_D^r - 2r + x_D < 1$, and since all variables are integers,

$$\varepsilon_D^r - 2r + x_D \leqslant 0 \tag{26}$$

And while for the second scenario shown in Fig. 6(c) (vertex I), where initial vertices C and D of r- and $(r + 1)$-circles are separated by an NBP vertex D, the first condition is sufficient, we can show that (26) still holds true. Indeed,

$$\varepsilon_I^r = \varepsilon(x_D - 1, y_D + 1, r) = \varepsilon_D^r + 2(y_D - x_D + 1) \tag{27}$$

$$(\varepsilon_I^r - 2r + x_I) = \varepsilon_D^r - 2r + x_I + 2(y_D - x_D + 1) \tag{28}$$

$$= (\varepsilon_D^r - 2r + x_D) + 2(y_D - x_D) + 1 \tag{29}$$

$$\leqslant 2(y_D - x_D) + 1 \tag{30}$$

In octant 1 $(y_I - x_I) \leqslant 1$, and then $(y_D - x_D) \leqslant -1$ and $2(y_D - x_D) + 1 < 0$. We thus showed that $\varepsilon_I^r - 2r + x_I < 0$.

2.5. Distance error in NBPs

Based on conditions (10) and (24), we can determine bounds for distance error at NBPs. Indeed, given that in octant 1, $r \leqslant \sqrt{2}x$, conditions (10) and (24) imply that

$$(1/\sqrt{2})r \leqslant \varepsilon_{NBP} \leqslant (2 - 1/\sqrt{2})r \tag{31}$$

or, roughly,

$$0.7r < \varepsilon_{NBP} < 1.3r \tag{32}$$

Based on (11) (Fig. 4), this guarantees that

$$0 < \delta_{NBP} < 1 \tag{33}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

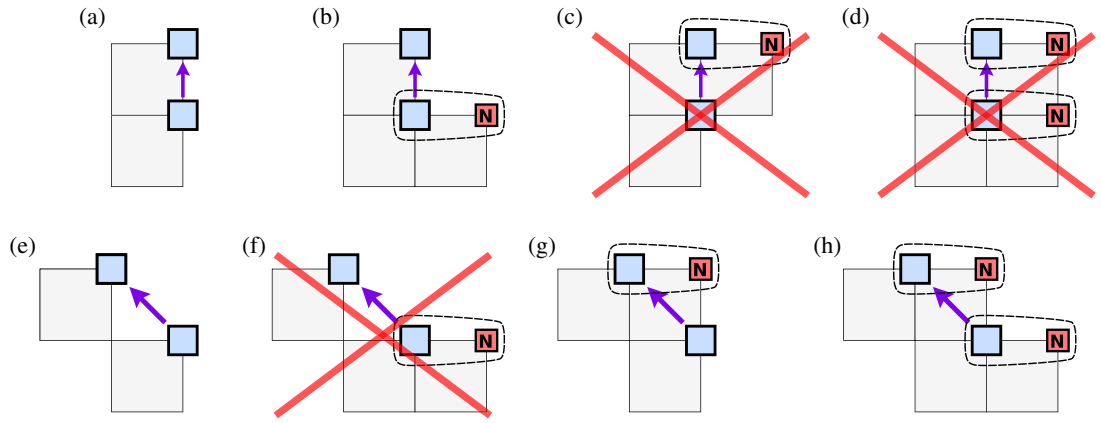

Fig. 7. Possible configurations of two sequential vertices/pairs while iterating through a Bresenham circle.

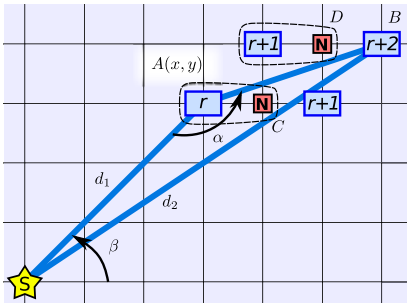

Fig. 8. An impossible configuration of two diagonal NBPs.

We can determine tighter bounds for distance error in NBPs. It's easy to verify that if $r \geqslant 3$, then $(1/\sqrt{2})r > 0.33^2 + (2r)(0.33)$. Thus

$$0.33^2 + (2r)(0.33) < \varepsilon_{NBP} < 0.65^2 + (2r)(0.65) \tag{34}$$

Based on the relation (11), we then conclude that

$$0.33 < \delta_{NBP} < 0.65 \tag{35}$$

The only NBP with $r < 3$ is (1, 1), where $\delta_{NBP} \approx 0.414$; thus, bounds (35) are correct for all NBPs.

2.6. Possible vertex configurations

In order to visit all vertices of the grid, it is necessary to develop a method of visiting NBPs while iterating through regular Bresenham vertices. The properties of the NBP formation process described above determine that NBPs cannot be horizontally or vertically adjacent to each other, and thus, every NBP can be *paired* with a regular vertex to the left from it (in octant 1) as shown in Fig. 5. And thus to visit all vertices of the grid, whenever there is a diagonal move (such as $D \to E$ in Fig. 2), an additional condition (24) has to be checked. If it's true, then vertex F is an NBP that also belongs to the Bresenham r-circle and has to be visited and assigned the distance.

We can also observe that among all imaginable configurations of two sequential vertices/pairs (Fig. 7), only the following five are valid: (a), (b), (e), (g), and (h). Indeed, (c) and (d) are impossible because NBPs can only be formed when there's a diagonal move while iterating a Bresenham circle (e.g. $A \to C$ in Fig. 6(a)). Configuration (f) is impossible because the diagonal move would result in an NBP to the right from the top vertex (Fig. 6(c)).

It can be shown that another configuration of diagonal NBP vertices is also impossible. Even though the current implementation of CWave does not rely on the following proposition, we present it and its proof here for the sake of completeness.

Proposition 2. *The diagonal configuration of NBPs in octant 1 shown in Fig. 8 is impossible.*

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

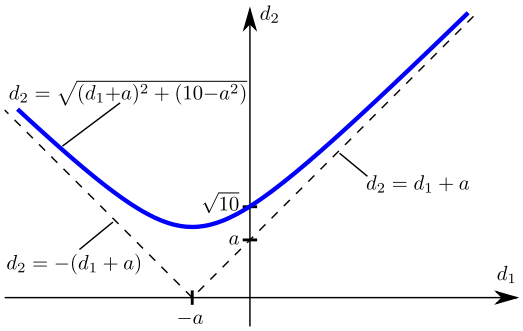

Fig. 9. Curve $d_2(d_1) = \sqrt{(d_1 + a)^2 + (10 - a^2)}$ and its asymptotes.

Proof. We will employ a proof by contradiction. Based on the law of cosines for $\triangle SAB$ (refer to Fig. 8),

$$d_2 = \sqrt{d_1^2 + |AB|^2 - 2d_1|AB|\cos\alpha} \tag{36}$$

$$= \sqrt{d_1^2 + 10 + 2\sqrt{10}d_1 \cos\left(\beta - \arctan\frac{1}{3}\right)} \tag{37}$$

$$= \sqrt{d_1^2 + 2d_1 a + 10} = \sqrt{(d_1 + a)^2 + (10 - a^2)} \tag{38}$$

where

$$a = \sqrt{10}\cos\left(\beta - \arctan\frac{1}{3}\right) \tag{39}$$

Figure 9 depicts the plot of $d_2(d_1)$ curve. Given the asymptotic properties of the curve, for all d_1:

$$d_2 - d_1 > a \tag{40}$$

$$r + 2 + \delta_2 - (r + \delta_1) > a \tag{41}$$

$$\delta_2 - \delta_1 > a - 2 \tag{42}$$

We can see from (39) that for $\beta \leqslant 2\arctan\frac{1}{3}$,

$$a \geqslant 3 \tag{43}$$

and thus from (42)

$$\delta_2 - \delta_1 > 1 \tag{44}$$

which contradicts the distance error boundaries in (12); thus, we only need to prove the statement for

$$\beta > 2\arctan\frac{1}{3} \tag{45}$$

First, let us show that the previous vertex for B must have been D_1, not B_1 (see Fig. 10(a)). Indeed, if the previous vertex is B_1 (refer to Fig. 10(b)), then D_1 is an NBP as well, and then the previous vertex for E is E_1, and then C_1 is an NBP. We can see that now we have a new pair of diagonal NBPs C_1 and D_1, which is located below C and D. We can continue this procedure further until we reach A_N for which $\beta \leqslant 2\arctan\frac{1}{3}$, but we showed that for such values of β the given configuration is impossible. Thus, based on condition (10) for vertex B, we can write

$$\varepsilon_B < x_B = x + 3 \tag{46}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

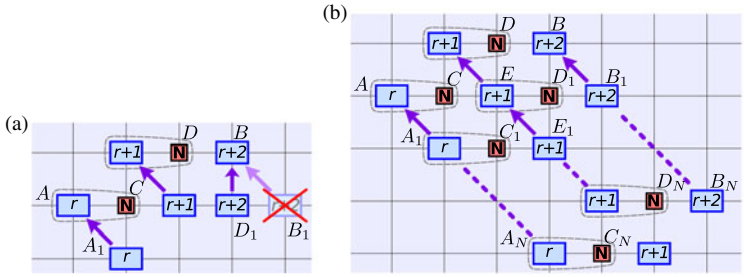

Fig. 10. Proving an impossible diagonal configuration of NBPs. (a) The previous vertex for B is D_1, not B_1; (b) if the previous vertex for B is B_1, then the diagonal NBP configuration has to repeat itself at vertices C_1 and D_1, and it can be continued indefinitely.

On the other hand, given (45),

$$x < r\cos\beta = \frac{1-\tan^2\left(\arctan\frac{1}{3}\right)}{1+\tan^2\left(\arctan\frac{1}{3}\right)}r = \frac{1-\frac{1}{9}}{1+\frac{1}{9}}r = 0.8r \tag{47}$$

and thus

$$\varepsilon_B < 0.8r + 3 \tag{48}$$

We can bring inequality (48) into a form suitable for estimating distance error δ_1 at vertex A (see (11)). Indeed, for $r \geqslant 142$, $0.8r + 3 < 0.41^2 + (2r)(0.41)$, and thus for such values of r,

$$\varepsilon_B < 0.8r + 3 < 0.41^2 + (2r)(0.41) \tag{49}$$

The last inequality gives us an upper boundary for δ_2:

$$\delta_2 < 0.41 \tag{50}$$

Now similarly we can find a lower boundary for δ_1. From condition (10) for vertex C:

$$\varepsilon_C \geqslant x_C = x + 1 \tag{51}$$

and thus

$$\varepsilon_A = \varepsilon_C - 2x_C + 1 = \varepsilon_C - 2x - 1 \geqslant -x > -0.8r \tag{52}$$

Again, for $r \geqslant 9$, $-0.8r > (-0.41)^2 + (2r)(-0.41)$, and thus

$$\delta_1 > -0.41 \tag{53}$$

On the other hand, even though the upper boundary for β in octant 1 can be slightly bigger than 45° (if NBPs are located on 45° diagonal), still for $r > 150$, $|SC| > 150 \Rightarrow \tan\beta < \frac{150}{149} \Rightarrow$

$$a > \sqrt{10}\cos\left(\arctan\frac{150}{149} - \arctan\frac{1}{3}\right) > 2.82 \tag{54}$$

and, thus, from (42),

$$\delta_2 - \delta_1 > 0.82 \tag{55}$$

Inequalities (55), (53), and (50) contradict each other; therefore, the proposition is proven for $r > 150$. For $r \leqslant 150$, the proposition can be verified by a computer program or manually.

2.7. Assigning distance values to vertices

Now that we are able to iterate through all vertices, we need to assign each vertex P a distance value $\bar{d}_P$. Clearly, the exact floating-point distance value is

$$d_P = \sqrt{x_P^2 + y_P^2} \tag{56}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

Since ε is already maintained for each vertex and r^2 can be calculated only once for each circle, the following expression is slightly faster to evaluate:

$$d_P = \sqrt{\varepsilon + r^2} \tag{57}$$

For integer-arithmetic-only implementation, we can trade off the square root for a small bounded error in each assignment. One approach is to define $\bar{d}_P = r$. In this case, however, given the distance error bounds (12) and (33), the true distance d_P will stay in a rather wide interval

$$\bar{d}_P - 0.5 < d_P < \bar{d}_P + 1 \tag{58}$$

which, however, does not increase with the growth of d_P.

We also developed another integer distance assignment method that allows to shrink the length of the interval to 0.5. It achieves three goals: it (1) allows fractional numbers in the integer variable $\bar{d}_P$ by using $2r$ instead of r, (2) adds an additional 0.5-offset if $d_P > r$, and (3) adds an additional 0.5-offset to NBP vertices:

$$\bar{d}_P = \begin{cases} \begin{cases} 2r+1 & \varepsilon_P > 0 \\ 2r & \varepsilon_P \leqslant 0 \end{cases} & \text{if } P \text{ is a regular vertex} \\ \begin{cases} 2r+2 & \varepsilon_P > r \\ 2r+1 & \varepsilon_P \leqslant r \end{cases} & \text{if } P \text{ is an NBP} \end{cases} \tag{59}$$

Overall, this assignment guarantees that

$$\frac{\bar{d}_P}{2} - 0.5 < d_P \leqslant \frac{\bar{d}_P}{2} \tag{60}$$

Indeed, from (11) (Fig. 4), given that ε_P and r are integers, the following relations follow:

$$\varepsilon_P > 0 \Rightarrow \delta_P > 0 \qquad \varepsilon_P \leqslant 0 \Rightarrow \delta_P \leqslant 0 \tag{61}$$

$$\varepsilon_P > r \Rightarrow \delta_P > 0.5 \qquad \varepsilon_P \leqslant r \Rightarrow \delta_P < 0.5 \tag{62}$$

When these inequalities are combined with (12) and (33), the inequality (60) becomes evident.

3. CWave on a Grid with Simply Connected Obstacles

By this point, we have established how CWave assigns distances in the special case of a free map. In this section, we will consider maps with *simply connected obstacles*, that is, maps where any two occupied cells can be connected by a path that never passes through a free cell. This topology guarantees that during CWave expansion, no vertex is visited more than once. We can, thus, defer the problem of wave merging to Section 4. First, however, we need to introduce some definitions.

3.1. Coordinate frames and definitions

Similarly to the Theta* approach, we add a single-occupied-cell wall around the perimeter of every map, as shown in Fig. 11, to eliminate the need for out-of-boundary checks. The bottom left *cell* has *absolute coordinates* [0, 0]. Vertex located at the left bottom corner of cell [x, y] has same coordinates [x, y]. Square brackets [.] are used for *absolute coordinates*, and parentheses (.) are used for *relative coordinates* (w.r.t. the source vertex). For example, vertex A in Fig. 11 has absolute coordinates [9, 5] and relative coordinates (with reference to source S) (7, 3).

We introduce the following definition of *visibility of vertex Y from vertex X* (Fig. 11).

Definition 1. *Vertex Y is NOT visible from vertex X if there is at least one point of the open line segment XY that (1) is also an inner point of any occupied cell OR (2) belongs to a line segment connecting centers of any two adjacent occupied cells. Otherwise, it is visible.*

Figure 11 demonstrates several examples. Here vertex A is not visible from S, because the line segment AS crosses an occupied cell [8, 4], and thus, there are inner points of AS that are also inner points of that cell. Vertex C is not visible, because vertex [8, 5] (an inner point of SC) belongs to the line segment connecting centers of two adjacent occupied cells: [7, 5] and [8, 4]. Similarly, vertex E is not visible, because vertex D is the inner point pf SE and belongs to the line segment connecting

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

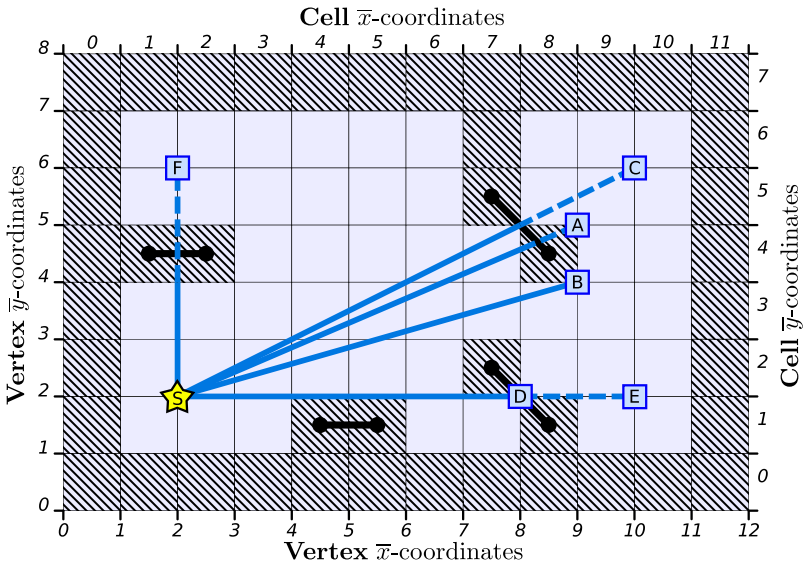

Fig. 11. Visibility of various vertices from vertex *S*. Visible vertices: *B* and *D*. Not-visible vertices: *A* (*SA* passes through inner points of cell [8, 4]), *C*, *E*, and *F* (crossing line segment connecting two adjacent occupied cells).

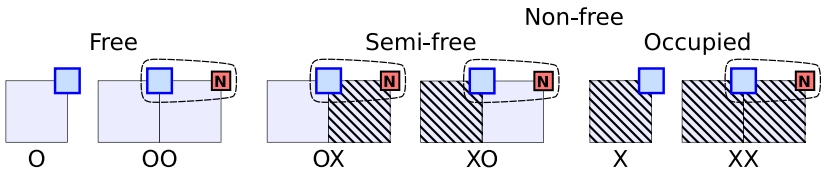

Fig. 12. Classification of grid cells/pairs occupancy patterns (*O* represents a free cell, *X* represents an occupied cell).

centers of two adjacent occupied cells: [7, 2] and [8, 1]. Note, however, that *D* itself is visible because *D* is not an *inner* point of line segment *SD*. Vertex *F* is not visible, because *SF* crosses line segment connecting adjacent occupied cells [1, 4] and [2, 4]. Finally vertex *B* is visible because inner points of *SB* do not overlap with any occupied cell or any line segment connecting two adjacent occupied cells.

We will say that *cell A corresponds to* vertex *B* if *B* is the furthermost corner of cell *A* (as measured from the source). In octant 1, thus, vertex *A* ($y > 0$) will not be visible if its corresponding cell is occupied. Overall, six occupancy patterns are possible. In Fig. 12, they are classified into *free*, *non-free*, *semi-free*, and *occupied*.

If all corresponding cells are free, we call the pattern *free*, otherwise the pattern is *non-free*. The *non-free* patterns are further divided into *occupied* (all corresponding cells are occupied), and *semi-free* (some corresponding cells are free, some are occupied). Note the pattern names comprised symbols *O* (designating a free cell) and *X* (designating an occupied cell).

3.2. Visibility cone

Consider a configuration shown in Fig. 13. Here a circular wave is expanding from source *S*, as occupied cells, represented by the shaded squares, block visibility of certain vertices on the map. The vertices that belong to the 2D cone formed between rays α and β are still visible from *S*, and MPC algorithm is still applicable, but it has to be bounded between α and β. We will call the 2D cone formed between α and β a *visibility cone*. The solid blue squares mark the outermost vertices of the cone whose visibility has to be checked. We will call this set of vertices *discrete closed boundaries*, where *closed* implies that vertices are permitted on the cone boundary rays.

Note that the shortest paths to other vertices on the map have to pass through either *A* or *B*, we can, thus, place new sources at those vertices. It is important to observe, however, that the upper discrete

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

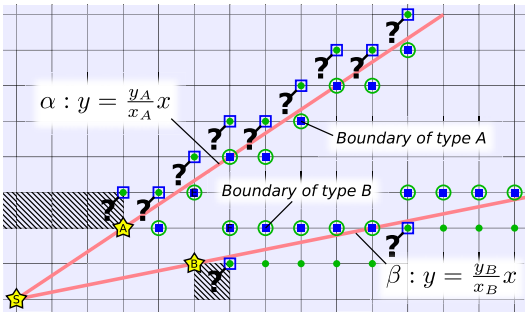

Fig. 13. An example visibility cone S, where S is the start source vertex; A and B are secondary sources; α and β are continuous boundaries of the cone S; solid blue squares are discrete closed boundaries of the cone S; circular green outlines are discrete open boundaries of cones A and B; cells marked with question marks can block parts of the visibility cone S, even though their corresponding vertices (blue square outlines) are not visible from S.

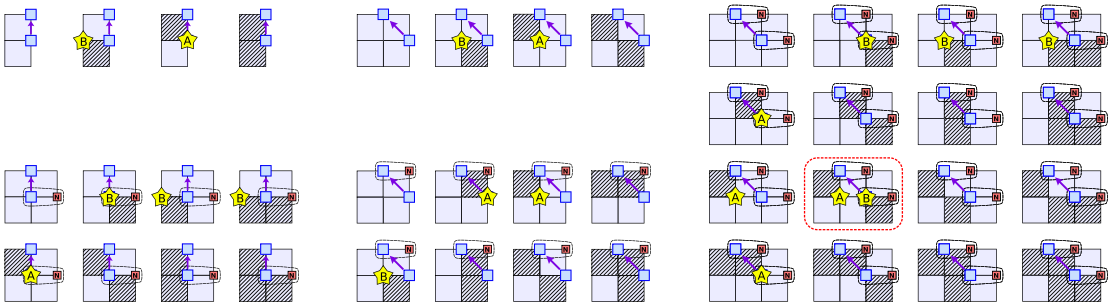

Fig. 14. All 40 possible combinations of two sequential configurations in octant 1. There is only one special case (labeled by a dashed rectangle) when two sources are added.

boundary of the new cone B (solid green circles in Fig. 13) is *open* (vertices that lie on β belong to cone S, not cone B). Same is true for the lower boundary of the new cone A.

3.3. Identifying corner vertices

In Section 2.6, we showed that there are only five valid configurations of vertices on two sequential iterations (Fig. 7). Given the six possible occupancy patterns in Fig. 12, we can identify 40 possible combinations (Fig. 14).

For all combinations, except for a special case of $OX \rightarrow XO$, a corner vertex appears only when a *non-free pattern* follows a *free pattern* or vice versa. In the first case, the new source will have a boundary of type A, whereas in the second case the new source will have a boundary of type B.

Combination $OX \rightarrow XO$ is classified as a special case (Fig. 15). Even though a *non-free pattern* XO follows another *non-free pattern* OX, a narrow visibility cone can still pass between the occupied cells. And in this case, two new sources are placed at A and B.

3.4. Iterating through boundary vertices

To iterate through boundary vertices (solid blue squares in Fig. 13), we developed a modified version of the Bresenham's line algorithm.[26] It needs only integer addition to operate. First, we will consider boundary of type A. At every step, the x-coordinate is incremented, but the y-coordinate is incremented only if

$$y + 1 \leqslant \frac{y_A}{x_A}(x + 1) \tag{63}$$

It can be noted that this expression has multiplication and division in it. If, however, we introduce the function

$$G(x, y) = (x+1)y_A - (y+1)x_A \tag{64}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

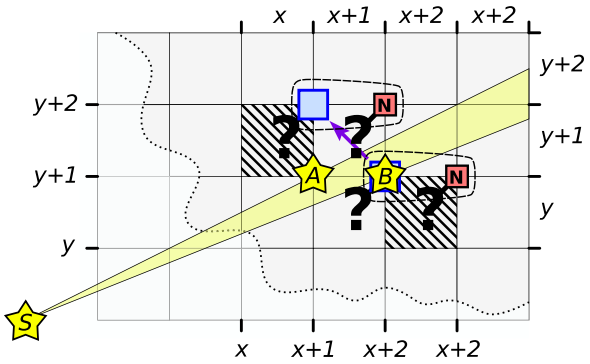

Fig. 15. Combination $OX \rightarrow XO$ allows a narrow visibility cone to pass between the occupied cells.

then the condition can be transformed into

$$G(x, y) \geqslant 0 \tag{65}$$

It is easy to observe that $G(x, y)$ can be calculated iteratively using only integer addition. Indeed, the initial value for G:

$$G(x_A, y_A) = y_A - x_A \tag{66}$$

and the update rules (depending on whether y is incremented or not) are

$$G(x+1, y) = G(x, y) + y_A \tag{67}$$

$$G(x+1, y+1) = G(x, y) + y_A - x_A \tag{68}$$

For type-B boundaries, y-coordinate is incremented if

$$y < \frac{y_B}{x_B}(x + 1) \tag{69}$$

or, if we introduce

$$G(x, y) = (x+1)y_B - yx_B - 1 \tag{70}$$

then the condition can be transformed into

$$G(x, y) \geqslant 0 \tag{71}$$

The initial value for G:

$$G(x_A, y_A) = y_B - 1 \tag{72}$$

and the update rules are

$$G(x+1, y) = G(x, y) + y_B \tag{73}$$

$$G(x+1, y+1) = G(x, y) + y_B - x_B \tag{74}$$

The boundaries, thus, can be iterated using only integer addition operation. Types A and B differ only in the initial value for G. Note that the value of the error function ε is propagated along the boundary vertices, again using only integer addition and bit-shifting:

$$\varepsilon(x+1, y, r+1) = \varepsilon(x, y, r) + 2(x - r) \tag{75}$$

$$\varepsilon(x+1, y+1, r+1) = \varepsilon(x, y, r) + 2(x + y - r) + 1 \tag{76}$$

These equations determine the initial value of ε for the MPC algorithm.

3.5. Determining arc endpoints from the boundary vertices

CWave iterates through MPC arcs and increments the radius at every step. In some cases, however, when moving from one boundary vertex to the next one, the radius increases by 2, not by 1. For

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

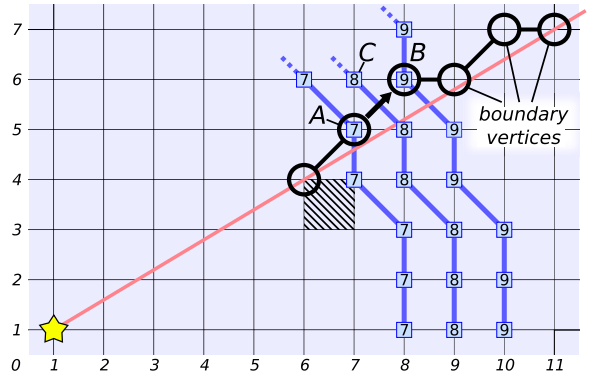

Fig. 16. Next vertex after A ($r = 7$) should be C ($r = 8$), not B ($r = 9$).

example, in Fig. 16, when moving from vertex A to vertex B, the radius changes from 7 to 9. In fact the correct starting vertex after A ($r = 7$) is C ($r = 8$), not B ($r = 9$). Such configurations are easy to detect, because for vertex B condition (10) will hold true. The arc endpoint vertex may also have an NBP vertex paired with it. That has to be checked with condition (24).

Another caveat is that even though the vertices marked with blue square outlines (not solid squares) in Fig. 13 are located outside of the visibility cone S, their corresponding cells (marked with question marks) can actually block visibility of some of the vertices inside the cone; thus, the occupancy of those cells has to be checked as well when calculating the arc endpoints.

3.6. Distance accuracy on a simply connected map

When any new source is added (see Section 3.3), its initial distance value $\bar{s}$ is taken from the integer distance previously assigned to the vertex by CWave. Given the distance error boundaries (60), each new source may, thus, introduce an accumulative distance error $|\delta| < 0.5$. That is the price of the purely integer arithmetic (addition and bit-shifting operations only) solution for the path planning problem. We will refer to such integer implementation of CWave as *CWaveInt*.

To eliminate the accumulative error, we developed the following modification of CWave (*CWaveFpuSrc*, where *Fpu* stands for *Floating-Point Unit*, *Src* stands for *source*). In addition to the integer source distance $\bar{s}$, for every source, we maintain a floating-point source distance s. It is initialized as

$$s = \sqrt{x_S^2 + y_S^2} + s_{parent} \tag{77}$$

but then is immediately rounded as

$$\bar{s} = round(2s) \tag{78}$$

The floating-point value s is used only when the source creates new sources at corner vertices (see Section 3.3). For basic distance assignment, only integer value $\bar{s}$ is employed. This approach allows every source to maintain the exact length of the shortest path to the start vertex (to the extent permitted by the machine epsilon), and, thus, eliminates the accumulative distance error.

We can calculate the total error of integer distances assigned by CWave in this case. First, from (78),

$$\frac{\bar{s}}{2} - 0.25 \leqslant s < \frac{\bar{s}}{2} + 0.25 \tag{79}$$

Now, combining this with (60), we will have

$$\frac{\bar{s} + \bar{d}}{2} - 0.75 < s + d < \frac{\bar{s} + \bar{d}}{2} + 0.25 \tag{80}$$

$$\frac{\bar{\rho}}{2} - 0.75 < \rho < \frac{\bar{\rho}}{2} + 0.25 \tag{81}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

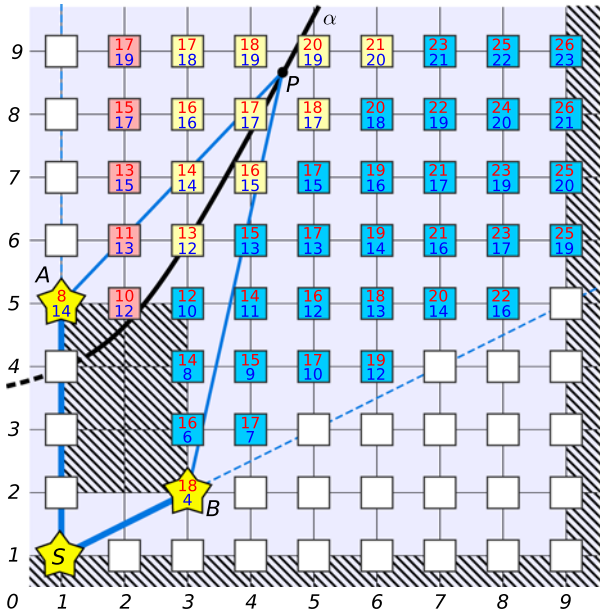

Fig. 17. A merge of Wave-*A* and Wave-*B*. Red and blue numbers designate CWave distance from the source *S* to the given vertex for paths passing through *A* and *B*, respectively. For pink and light blue vertices, the integer criterion (85) is sufficient to determine which path from *S* is shorter: through *A* (pink) or through *B* (light blue). For yellow vertices, the criterion is not sufficient, they form an overlapping area.

where $\rho = s + d$ is the exact length of the shortest path from the start source to a given vertex, and $\overline{\rho} = \overline{s} + \overline{d}$ is the corresponding CWave integer distance value.

Given the minimal computational overhead of this modification ((77) is calculated only when a new source is created), it is used in the further analysis.

3.7. Path extraction on simply connected maps

Path extraction is the process of determining the shortest path (not just the length of the path) from the initial source vertex to any other vertex on the map. When there is a need for path extraction, two additional data structures are maintained: a 1D ordered *sources_array* that accumulates coordinates of all source vertices and a 2D *track_map* that, for every vertex on the map, stores the index of the source that assigned the vertex its distance value. Once the distance map is calculated, a path from any given vertex $P(x_P, y_P)$ can be recursively extracted: read P's nearest source id $S1_{id} = track\_map[P.x, P.y]$, read $S1$'s coordinates: $S1 = sources\_array[S1_{id}]$, read $S1$'s nearest source id $S2_{id} = track\_map[S1.x, S1.y]$, and so on until the start source vertex is reached. The modification of CWave that maintains *sources_array* and *track_map* is referred to as *CWaveTrack*.

In future, gradient methods can be explored for path extraction directly from the distance map. Such approach would render *sources_array* and *track_map* unnecessary, but, given the integer nature of CWave distance map, it may encounter certain challenges.

4. CWave on a Generic Map

On a generic map, obstacles are not necessarily *simply connected*, and thus waves from two or more sources may eventually "meet" each other. This section discusses the case of merging waves for a realistic 2D map in which we release the assumption of simply connected obstacles.

4.1. Boundary between merging waves in continuous space

Figure 17 illustrates the scenario in which two waves are merging. Here *S* is a parent source for secondary sources *A* and *B* which generate wave-*A* (red vertices) and wave-*B* (blue vertices), respectively. These two waves merge along a certain boundary α. In a continuous 2D space, we can define α as a set of points *P* for which the lengths of the polygonal paths *SBP* and *SAP* are equal.

Proposition 3. *α is a hyperbola.*

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

Proof. By definition of α,

$$|SB| + |BP| = |SA| + |AP| \tag{82}$$

$$|AP| - |BP| = |SB| - |SA| \tag{83}$$

$$|AP| - |BP| = const \tag{84}$$

The last equation is one of the standard definitions of a hyperbola.

For vertices located on one side of the hyperbola, the shortest path to S passes through A, and for vertices on the other side, the shortest path passes through B.

4.2. Integer criterion

Let us imagine, what will happen if we run the current implementation of CWave on a map with non-simply connected obstacles as shown in Fig. 17. Wave-A and Wave-B will eventually *overlap*; that is, some of the vertices on the map will be visited by both of the waves. Moreover each wave will independently place a new source at the top right corner of the obstacle (vertex [3, 5]) and then at the next corner, and so on. The waves will "wrap" around the obstacle infinitely.

Visiting the same vertex multiple times, while somewhat degrading the performance, will not affect the accuracy, if the new distance value assigned to the vertex is smaller than the previous value, but it's critical to prevent infinite loops.

The following proposition serves as an integer criterion to determine when Wave-B should stop penetrating Wave-A:

Proposition 4. *If for a given point P*

$$\overline{\rho}^A \leqslant \overline{\rho}^B - 2 \tag{85}$$

then

$$\rho^A < \rho^B \tag{86}$$

where $\overline{\rho}^A$ and $\overline{\rho}^B$ are integer distance values assigned to P by sources A and B, respectively, and ρ^A and ρ^B are exact lengths of the paths SAP and SBP.

Proof. From (81),

$$\rho^A - 0.75 < \frac{\overline{\rho}^A}{2}; \qquad \frac{\overline{\rho}^B}{2} < \rho^B + 0.25 \tag{87}$$

Substitution of (87) into (85) proves the proposition.

The practical interpretation of this proposition is that if source B tries to assign value $\overline{\rho}^B$ to a certain vertex P, whereas for the current value $\overline{\rho}^A$ of P, inequality (85) holds true, then we can be certain that *SAP* is shorter than *SBP*. That means that we can treat vertex P as if we reached an occupied cell and limit the visibility cone angular range of the source A to P. That also implies that A should not place a new source at vertex P.

For yellow vertices (Fig. 17), however, condition (85) is false, which means that the integer values $\overline{\rho}^A$ and $\overline{\rho}^B$ are not sufficient to determine which path is shorter for a given vertex P: *SAP* or *SBP*. The two waves, thus, *overlap* at the yellow vertices. If there is an occupied cell in that area, both waves may place a new source at the same corner.

4.3. Distance accuracy on a generic map

Proposition 5. *Let $\overline{\rho}^A$ be the integer distance value assigned by CWave to vertex P by source A, and $\overline{\rho}^B$ be the integer distance value that is later attempted to be assigned by CWave to vertex P by source B. If we choose to update the distance value at P with $\overline{\rho}^B$ only when*

$$\overline{\rho}^B < \overline{\rho}^A \tag{88}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

then

$$\left|\rho - \frac{\overline{\rho}}{2}\right| < 0.75 \tag{89}$$

where ρ is the length of the true shortest path to P, and $\overline{\rho}$ is the integer distance value assigned by CWave.

Proof. Given the previous definitions of $\overline{\rho}^A$ and $\overline{\rho}^B$, the following cases are possible:

(1) $\overline{\rho}^B \leqslant \overline{\rho}^A - 2$ (based on condition (88), path via B is chosen).
Proposition 4 implies that, in this case, $\rho^B < \rho^A$, and thus path through vertex B is chosen correctly. This means that the error boundaries for $\overline{\rho} = \overline{\rho}^B$ can be calculated using (81). The interval defined by (81) is contained in the interval defined by (89).
(2) $\overline{\rho}^B = \overline{\rho}^A - 1$ (based on condition (88), path via B is chosen).
In this case, two sub-cases are possible:
(a) $\rho^B \leqslant \rho^A$: path via B is chosen correctly, thus, again, error boundaries (81) are true;
(b) $\rho^B > \rho^A$: path via B is chosen incorrectly, the length of the shortest path to P is $\rho = \rho^A$, but, for ρ^A, error boundaries (81) are still true:

$$\frac{\overline{\rho}^A}{2} - 0.75 < \rho^A < \frac{\overline{\rho}^A}{2} + 0.25 \tag{90}$$

And, thus, after substituting $\overline{\rho}^B = \overline{\rho}^A - 1$, we will have

$$\frac{\overline{\rho}^B}{2} - 0.25 < \rho^A < \frac{\overline{\rho}^B}{2} + 0.75 \tag{91}$$

$$\frac{\overline{\rho}}{2} - 0.25 < \rho < \frac{\overline{\rho}}{2} + 0.75 \tag{92}$$

This interval is also contained in 89.
(1) $\overline{\rho}^B = \overline{\rho}^A$: (based on condition (88), path via A is chosen):
Since $\overline{\rho}^B = \overline{\rho}^A$, error boundaries (81) will be correct no matter which real distance ρ^A or ρ^B is shorter.
(2) $\overline{\rho}^B = \overline{\rho}^A + 1$ (based on condition (88), path via A is chosen):
This case is symmetric to case (2)—that is, it can be rewritten as $\overline{\rho}^A = \overline{\rho}^B - 1$.
(3) $\overline{\rho}^B \geqslant \overline{\rho}^A + 2$ (based on condition (88), path via A is chosen):
This case is symmetric to case (1)—that is, it can be rewritten as $\overline{\rho}^A \leqslant \overline{\rho}^B - 2$.

Proposition 5 provides distance error boundaries for a generic map.

4.4. Wave merge with floating-point criterion

If path tracking (see Section 3.7) is enabled and floating-point operations on a given platform are cheap, then another modification of CWave (*CWaveFpuMerge*) can be more preferable on certain types of maps. Indeed, when a wave-merge is detected at point P, but Proposition 4 does not hold, we can utilize path tracking to calculate exact floating-point distances ρ^A and ρ^B:

$$\rho^A = s_A + \sqrt{(x_P - x_A)^2 + (y_P - y_A)^2} \tag{93}$$

$$\rho^B = s_B + \sqrt{(x_P - x_B)^2 + (y_P - y_B)^2} \tag{94}$$

Now, if $\rho^B \geqslant \rho^A$, then wave B penetrated wave A too much, and vertex P value should not be reassigned.

Even though this criterion requires additional floating-point calculations (including two square roots), it eliminates the overlapping area, and thus, on certain types of maps where overlapping creates a significant overhead, it might be faster than *CWaveFpuSrc* or even *CWaveInt*. Additionally,

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

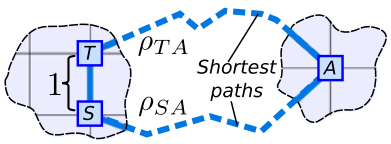

Fig. 18. Triangle inequality used for the shake test: $|\rho_{SA} - \rho_{TA}| \leqslant 1$.

by making an exact distance comparison (permitted by the machine epsilon) at all merge vertices, *CWaveFpuMerge* is able to find optimal paths. The exact length of the optimal path can be calculated as discussed is Section 6.2.

5. Performance Tests

The key performance characteristics of a path planning algorithm are accuracy and speed. They are discussed in the Sections 5.1 and 5.2, respectively.

5.1. Verification and accuracy

The implementation of the CWave algorithm is not a trivial task, due to numerous special cases that are easy to overlook (vertices on 45°-diagonals, cones that are narrower than a single cell, NBPs at non-marked vertices, vertices with $y = 0$). To achieve a desired level of reliability, several verification methods were developed. Let us first introduce a definition.

Definition 2. *If a vertex is surrounded by four occupied cells, it is called a blocked vertex. Otherwise, it is a non-blocked vertex.*

In the order of decreasing speed and increasing accuracy, the developed tests are the following:

(1) *Fill test*, for a set of simply connected maps, runs CWave from every non-blocked vertex on the map and verifies on the fly that, during each run, none of the vertices is visited more than once. At the end of each run, it checks that all non-blocked vertices on the map are assigned a distance value. This test is fast, but does not check for distance errors.
(2) *Shake test*, for a set of generic maps, indirectly verifies the distance accuracy by checking the triangle inequality. Indeed, for the lengths ρ_{SA} and ρ_{TA} (Fig. 18) of the two shortest paths calculated from adjacent vertices S and T to any vertex A, inequality $|\rho_{SA} - \rho_{TA}| \leqslant 1$ should hold true. Given the error bounds (89), this inequality implies

$$|\overline{\rho}_{SA} - \overline{\rho}_{TA}| < 5 \tag{95}$$

Shake test verifies that (95) is true for all non-blocked starting vertices S on the map and all non-blocked A. This self-test is several times slower than *Fill test*, but it catches accumulative distance errors.
(3) *Accuracy test* checks that error bounds (89) are correct by comparing the CWaves distances to the those calculated by ANYA,[27,28] an optimal any-angle path planning algorithm. Given that ANYA is designed for point-to-point path planning, a complete all-to-all distance check is very slow (for map in Fig. 19(h) it takes about 10 h).

5.2. Speed comparison to other algorithms

In ref. [17], a comprehensive comparative analysis is presented for most popular any-angle path planning algorithms: A* (8-connected, not any-angle), Theta*, Lazy Theta*, Block A*, Field A*, ANYA, and Any-Angle Subgoal Graphs. In particular, their speed and accuracy when solving point-to-point path planning problems are compared. We expanded on their test framework: integrated CWave into the test and adapted Theta*, Lazy Theta*, Field A*, ANYA, Block A*, and A* for single-source path planning by stopping the search only when all vertices are visited and setting heuristics to 0. Additionally, in case of ANYA, the path length to each vertex P in every interval was calculated according to

$$\rho(P) = g(R) + d(R, P) \tag{96}$$

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

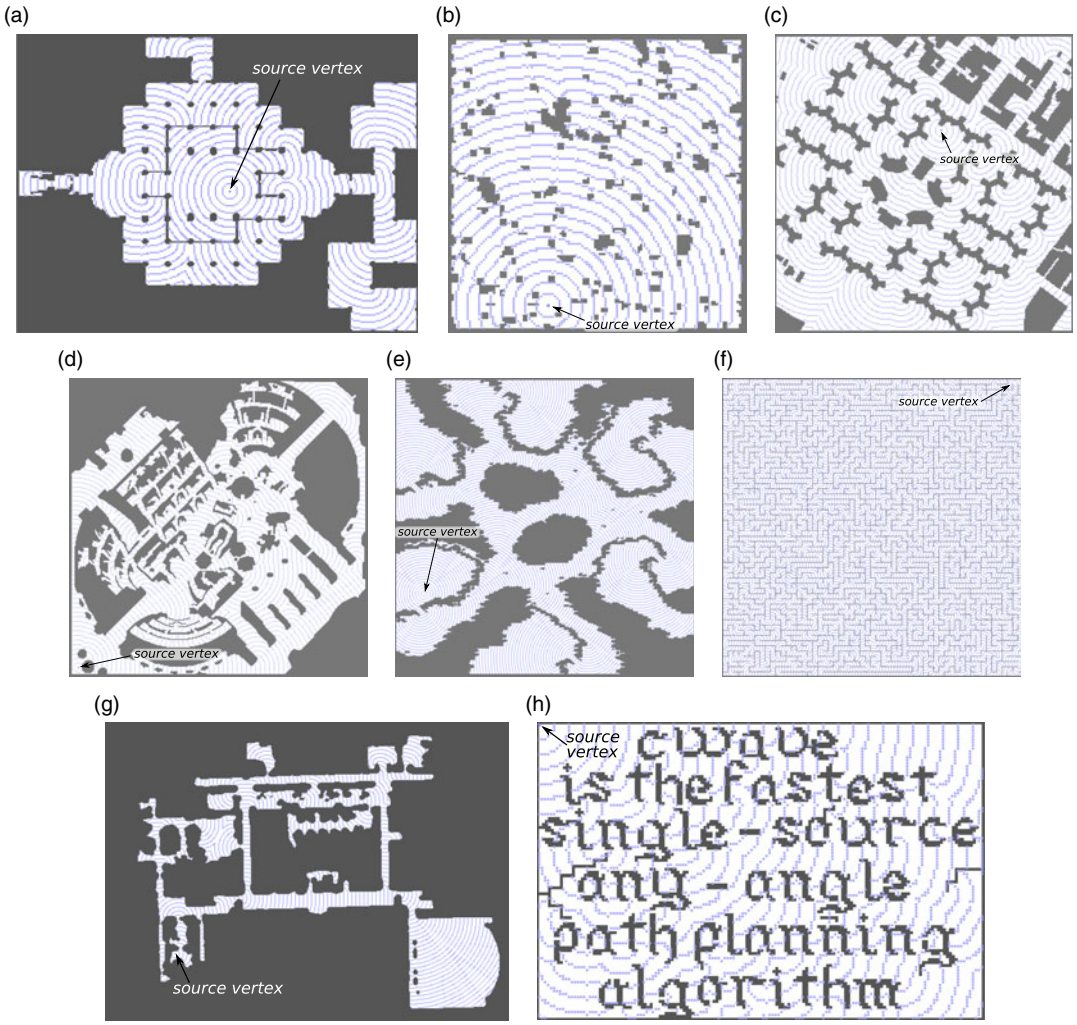

Fig. 19. Test maps except (g) and (h) are adopted from ref. [29]. The blue curves are equidistant from the given source vertex as calculated by CWave. They visualize a solution of one path planning problem for each map. (a) *arena2*: 281 × 209, 1000 problems, (b) *spots*: 128 × 128, 430 problems, (c) *nyc*: 256 × 256, 400 problems, (d) *baldurs*: 256 × 256, 270 problems, (e) *conquest*: 512 × 512, 200 problems, (f) *maze*: 512 × 512, 150 problems, (g) *aklabs*: 500 × 370, 800 problems, and (h) *cwave*: 150 × 100, 900 problems.

where $g(R)$ is the length of the optimal path from the start source vertex to the current root,[27] and $d(R, P)$ is the straight line distance from the root to the given vertex P that belongs to the current interval. Some of the vertices were updated multiple times, but only the minimum value was kept.

The framework measures the time required by each algorithm to solve a set of path planning problems on a given map. The tested maps are shown in Fig. 19, where the blue "curves" designate equidistant lines as calculated by CWave algorithm (distance between the curves is equal to 5 cell widths).

The tests were executed on Intel(R) Core(TM) i7-3610QM CPU @ 2.30 GHz (10 runs for each map). The results (Fig. 20) demonstrate that on all maps except *spots* and *cwave*, CWave performed faster than all other algorithms. The highest performance advantage is achieved on *conquest.map*: 16 times faster than Theta*, 8 times faster than Lazy Theta, and 2.5 times faster than the nearest competitor Block A*. Slightly worse, but similar results, are demonstrated on an inflated map of a real building *aklabs*. On the other hand, on maps with a higher number of isolated obstacles (*cwave.map* and even more so on *spots.map*), CWave in its basic implementation does not perform as competitively: while still being faster than all other tested algorithms, on *cwave.map* it yeilds to Block A* by 42%. On *spots.map*, Block A* beats CWave in speed fourfold (this is significantly improved in the following subsections). We believe that the primary reason of a comparatively poor performance

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

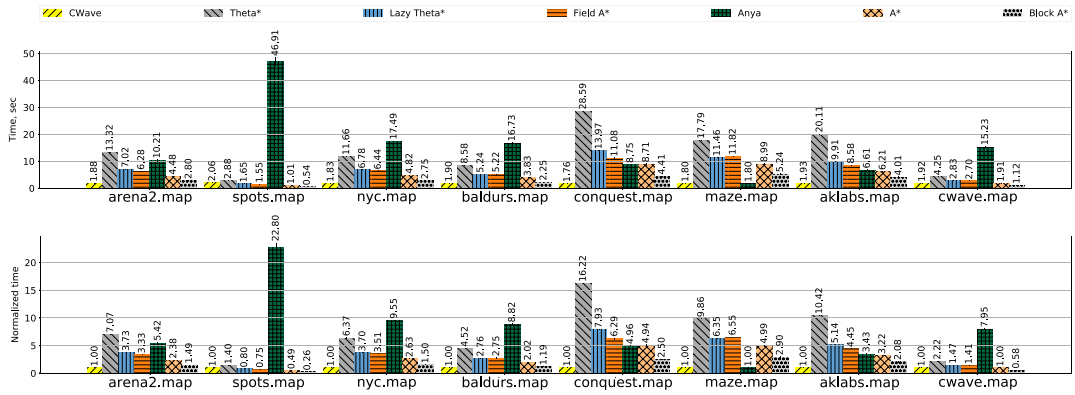

Fig. 20. Performance comparison of single-source path planning algorithms: measured in seconds (top), normalized by the average duration of *CWave* (bottom).

on such maps is that the main CWave performance advantage comes from representing the wave front as a set of discrete arcs and lines (rather than individual vertices); thus, if the number of cones grows too high, the efficiency of such representation drops. Maps with many isolated obstacles result in major wave overlappings and a high number of secondary waves which leads to the performance degradation. This hypothesis is further supported by the experimental results with multithreaded and modified implementations of CWave, discussed in the following. We believe that single-source ANYA suffers from a similar problem of unnecessary wave overlapping; however since it does not implement any wave merging mechanism, its speed is affected much worse: in the worst case of *spots.map*, it's almost 90 times slower than Block A*.

5.3. Parallelization

Most modern computers have several cores (CPUs) and some are even equipped with GPUs. A parallelized algorithm can distribute the computational load between multiple cores effectively decreasing the computational time.

One way to parallelize CWave is to distribute the wave sources available for expansion at every step between multiple threads running simultaneously. The effectiveness of such parallelization is highly dependent on the map. Indeed, a map without obstacles will never have more than one source, and, thus, only one thread will be doing useful work.

An important aspect of this parallelization scheme is the distance equivalence between the threads. In our current implementation, we chose to synchronize threads at every distance step $(d, d+2, \ldots, d+2k)$. This allows to keep frontier at the same distance and expand the wave at constant velocity. The practical implementation of such parallelization revealed a high degree of contention between the threads, rendering the usage of traditional thread synchronization means (semaphores, mutexes, and even spinlocks) ineffective: performance tests showed that all multithreaded implementations using these synchronization tools were slower than a single-threaded implementation. That can be explained by the fact that it is not only the set of sources that is shared between the threads, but also map vertices as well. A significant performance improvement, however, has been achieved by resorting to C++11 atomic variables and atomic CAS (compare and swap) operation on map vertices.[30]

The results of the performance tests comparing N-threaded implementation *CWaveN* with the single-threaded implementation *CWave* are presented in Fig. 21. *CWaveFpuSrc* was utilized as the basis for the test.

Among the expected results, we can observe that *CWave1* (a multithreaded implementation running a single thread) is slightly slower than *CWave* (the single-threaded implementation), due to the overhead of atomic variables. When the number of threads reaches the number of simultaneous threads supported by the given CPU (*CWave8*), the performance noticeably decreases. We can also observe that more intricate maps (resulting in more source vertices) show a better improvement from parallelization: in the best case on *spots.map* 8-threaded implementation,

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

Table II. Summary of CWave modifications.

Modification	Description	FPU operations	Distance accuracy
CWaveInt	Pure integer implementation using only addition and bit shifting operations	None	Accumulative error $\lvert\Delta d_k\rvert \leqslant 0.5$ at every turn
CWaveFpuSrc	Calculates floating-point distance value at corner vertices (Section 3.6)	At corner vertices	Non-accumulative error $\lvert\Delta d\rvert < 0.75$
CWaveTrack	Same as *CWaveFpuSrc*, but maintains additional data for path extraction (Section 3.7)	At corner vertices	Non-accumulative error $\lvert\Delta d\rvert < 0.75$
CWaveFpuMerge	Same as *CWaveTrack*, but uses a floating-point criterion at wave merge vertices (Section 4.4)	At corner and merge vertices	No distance error (optimal)

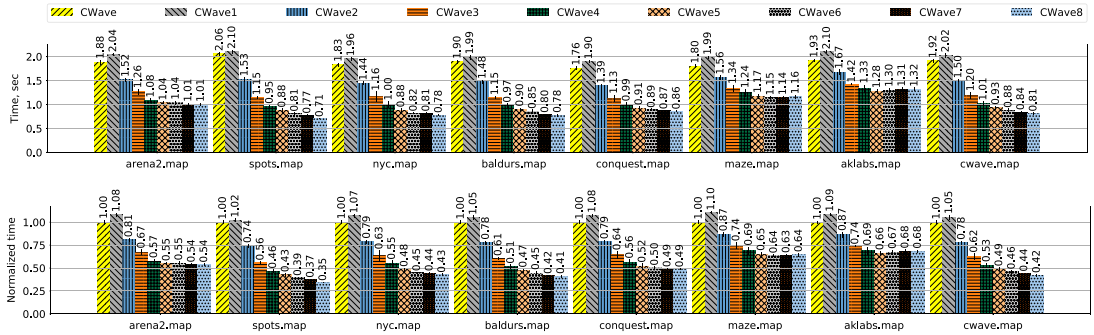

Fig. 21. Performance of 1- to 8-threaded implementations (*CWave1* to *CWave8*) compared to the single threaded implementation (*CWave*): measured in seconds (top), normalized by the average duration of the single-threaded implementation of *CWave* (bottom).

CWave8 is almost 3 times faster than the single-threaded implementation *CWave*, whereas the best multithreaded implementation *CWave5* on a topologically simpler map *aklabs.map* is just 50% faster than the single-threaded implementation. This supports the hypothesis that excessive branching that occurs on maps with a high number of isolated obstacles results in performance degradation which can be significantly compensated by multithreading. Indeed, CWave8 on *spots.map* despite major wave overlapping approaches the performance of Block A*. This is further improved in the next subsection.

As we can see, this approach to parallelization, while noticeably improving the performance, does not make it N-times faster (where N is the number of threads) than the single-threaded implementation. When the wave frontier is not required to stay at the same distance, a parallelized implementation with less frequent synchronization between the threads might yield a better performance increase.

From a practical point of view, for the tasks where distance maps need to be calculated from multiple sources, it might be more effective (time-wise) to run simultaneously N independent algorithms, one for each start source, rather than parallelize each run. This will almost completely eliminate the contention between the threads, but, of course, would require N-times more RAM.

5.4. Speed of CWave modifications

In this section, we compare the speed of the four modifications of CWave discussed so far: *CWaveInt*, *CWaveFpuSrc*, *CWaveTrack*, and *CWaveFpuMerge*. The properties of these modifications are summarized in Table II. Their speed has been measured using the same test setup as discussed in Section 5.2. The results (Figure 22) expectedly show that *CWaveFpuSrc* is just a little bit slower than *CWaveInt*, because the additional usage of floating point operations in *CWaveFpuSrc* is minimal.

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

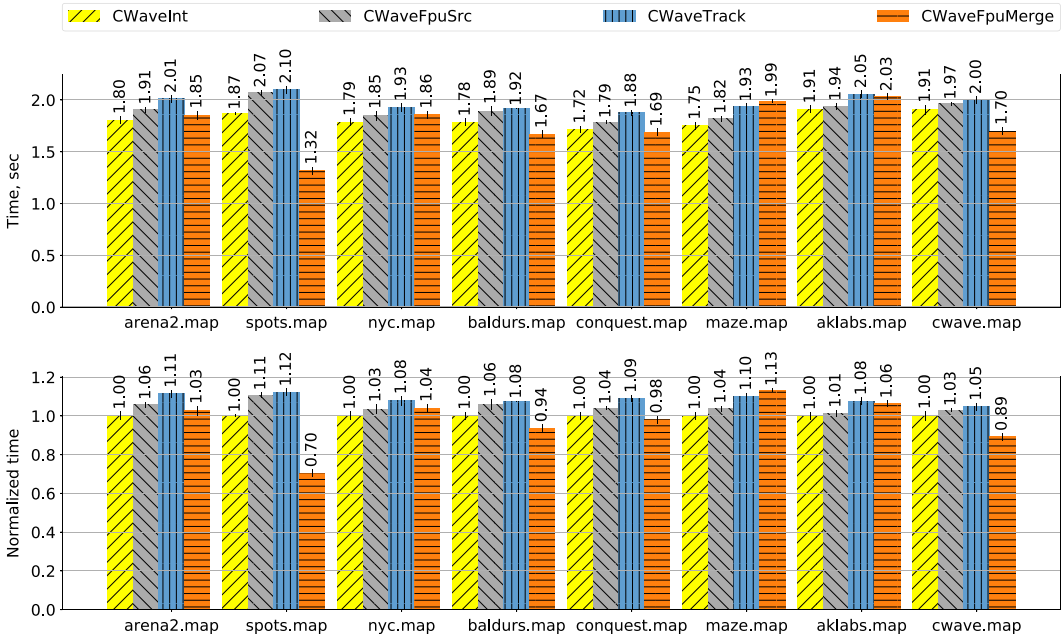

Fig. 22. Performance comparison of various modifications of CWave: measured in seconds (top), normalized by the average duration of *CWaveInt* (bottom).

Predictably, maintaining data for path extraction increases the runtime of *CWaveTrack* (as compared to *CWaveFpuSrc*) by 5–11%. What is interesting to observe is that on maps with higher number of isolated obstacles, such as *cwave.map* and *spots.map*, where more wave merging occurs, *CWaveFpuMerge* is noticeably faster than *CWaveFpuSrc*. Indeed, on *spots.map*, we gain an additional 30% of runtime reduction. Thus, even on this most challenging map for CWave, in combination with multithreading, CWaveFpuMerge may perform as fast as its closest "competitor" Block A*.

This supports our hypothesis that, on some maps, eliminating wave overlapping can increase the speed of the algorithm, despite the need to maintain additional tracking data and execute floating-point calculations at merge vertices.

6. Discussion and Future Work

In this section, we will discuss certain advantages and drawbacks of CWave, as well as the potential directions for future work.

6.1. Implementation

The main drawback of CWave is its implementation complexity which resulted in about 1500 lines of C++ code. It required very rigorous and time-consuming test process to debug. It is very likely that certain aspects of the algorithm/implementation can be generalized and simplified.

However, CWave has the main advantage of having superior performance over its predecessors. Given that CWave, in its most basic form *CWaveInt*, requires only integer addition and bit shifting (multiplication by two), it can be ported to low-cost embedded platforms that lack support for floating-point operations, for example, those used in swarm robotics.

The high-level pseudo-code of CWave is presented in Algorithm 1, where *GetNextVx*(*src*, *cur_vx*) iterates through vertices of a Bresenham arc as discussed in Section 2.1, *ExtendBdry*(*boundary*) iterates through vertices along boundary lines as discussed in Section 3.4; *CheckOccupancyPattern*(*vx*) checks for the occupancy of the cells corresponding to vertex *vx* and its NBP if it is present, assigns distance values to the vertices in *dist_map* (Section 2.7), and returns one of the patterns from Fig. 12; *ProcessConfiguration*(*prev_ptrrn*, *new_pttrn*) identifies one of the 40 configurations shown in Fig. 14, and, if corners are detected, updates boundary lines (may create new cones) and places new sources at the corners (adds them to *new_sources* array); and *ProcessStartVertex*(.) and *ProcessEndVertex*(.) can result in the same actions as *ProcessConfiguration*(.).

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

Algorithm 1 High-level CWave pseudo-code

```
Input: map (2D grid modeling the map), init_vx (initial source vertex)
Output: dist_map (2D array with distance value for every vertex)
Initialization
create source object from init_vx, place it into array sources
front_dist ← 2
while sources is not empty do
    for src in sources do
        src.dist ← src.dist + 2
        src.radius ← src.radius + 1
        // Source maintains a set of cones
        for cone in src.cones do
            start_vx ← ExtendBdry(cone.start_boundary)
            end_vx ← ExtendBdry(cone.end_boundary)
            prev_pttrn ← ProcessStartVertex(start_vx)
            // Iterate Bresenham arc vertices between start & end
               boundaries:
            cur_vx ← start_vx
            while cur_vx != end_vx do
                cur_vx ← GetNextVx(src, cur_vx)
                pttrn ← CheckOccupancyPattern(cur_vx)
                ProcessConfiguration(prev_pttrn, pttrn)
                prev_pttrn ← pttrn
            end
            ProcessEndVertex(end_vx)
        end
    end
    merge new_sources into sources
    from sources delete sources without cones
    front_dist ← front_dist + 2
end
```

6.2. Vertex exact floating-point distance calculation

Note that after the CWave integer distance values are calculated, the true floating-point distance can be extracted for any vertex P if needed. First, P's parent source needs to be identified (see Section 3.7). The source exact floating-point distance to the start source is already known to be s. Then using relative coordinates of P the exact floating-point distance d from P to parent source is calculated. Finally, $s + d$ gives the exact length of the shortest path from the start source vertex to P.

6.3. Reduction of thread contention in the multithreaded implementation

As discussed, the performance improvement of the current multithreaded implementation of CWave is pretty far from increasing proportionally to the number of threads. Certain changes to the design, however, are likely to make the multithreaded implementation more efficient.

For example, instead of distributing sources between the threads, individual cones can be distributed. Synchronizing threads less frequently would reduce thread contention, the key factor of performance degradation. Currently the threads are synchronized such as to keep the wave front at approximately the same distance, which is good to minimize the wave overlaps. If we allow threads to synchronize less frequently, some redundant occasional wave overlaps will be more likely, but at the same time there is a good chance that the reduced thread contention will result in an overall significant performance increase. These are still open directions for research.

6.4. Improving wave merges

In Section 4, we showed that the current implementation of CWave can generate multiple waves that overlap (i.e. they visit the same vertices multiple times). Even duplicate source vertices with slightly different distances can be created. This drawback can potentially be mitigated by not allowing source

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

duplicates. Indeed, if among all sources there are two or more sources with the same coordinates, one of them will have its floating-point distance s not greater than that of all others. Only that source can thus be kept, and others can be deleted.

An alternative solution to the problem of overlapping is to add a hyperbolic boundary to the overlapping waves, but that might not be as easy to construct using simple arithmetics. However, there is a certain chance that in many cases that boundary can be very well approximated by a straight line.

6.5. Performance tests on various classes of maps

An important direction for further research is a comprehensive comparison of CWave performance on different types of maps. Several classes of maps can be defined based on their resolution, obstacle density, obstacle size, and connectivity. Comparing CWave performance on these types of maps versus the performance of other single-source any-angle path planning algorithms can reveal other stronger and weaker sides of CWave.

6.6. Applications

As a fast single-source path planning algorithm on a grid, CWave may find its use in various robotic applications. In fact, it's not simply an algorithm, but is rather an efficient method of equidistant wave propagation on a grid. In this subsection, we will outline how CWave was utilized to enable robot navigation via an LTI and discuss other possible applications of the method.

As it was mentioned in Section 1, the development of CWave was motivated by a practical problem of robot navigation via LTIs, such as a brain–computer interface, facial expression control.[1] In ref. [2], we present the development of such system, as well as the results of simulation and real robot experiments.

To summarize, CWave was integrated with Robot Operating System (ROS)[31] and used internally in several critical components of the navigation system: (1) By limiting the maximum distance in CWave, we were able to efficiently calculate the robot reachability area online. On a grid map, the reachability area is a set of vertices that can be reached by a robot within the next T seconds from its current position. While the reachability area of a holonomic robot on an empty map is simply a disk, when obstacles are present, it has to be calculated using a singles-source path planning algorithm. (2) CWave was also employed to find intermediate destination points. The single-source distances calculated with CWave were weighted by the inferred probabilities of the destinations. The result was a probabilistic time-to-destination function. One of the navigation policies was to find a vertex in the reachability area that would minimize this function and use these points iteratively as a waypoint destinations until the intended destination is inferred with sufficient confidence. This method significantly reduced the total navigation time. (3) CWave played a critical role in the inference process as well. LTIs are typically limited to a small number of possible commands, whereas the total number of grid vertices is tens of thousands. As part of the iterative inference process, we had to repeatedly divide the grid map into four areas and let the operator select the desired one. CWave was the basis for two map segmentation policies: in one of them, the boundaries between the regions were aligned with equidistant curves (as measured from a given source vertex), and in the other one, they were aligned with extremals (shortest paths). The details of these implementations are out of the scope of this paper and can be found in refs. [2] and [32].

Navigation via LTI is not the only robotic application where the destination point is unknown. In fact, any intelligent robot is expected to understand *high-level* tasks and, thus, should be able to rationalize about its goal position rather than take it directly from the operator. In this sense, fast single-source path planning can efficiently provide information about all possible navigation scenarios, and may serve as the basis for an intelligent robot navigation strategy. For example, in robot exploration tasks, various areas of the map can be prioritized based on some external data. By combining this knowledge with the distance information calculated by CWave, the robot may develop an effective exploration method. Another interesting application is in multi-robot navigation. For example, in a real-time escape-and-pursue multi-robot game, several "predator" robots try to "catch" a "prey" robot. All players in this game will greatly benefit from using CWave to pre-calculate possible navigation scenarios for all other robots and find an optimal navigation strategy either to pursue or to escape.

https://doi.org/10.1017/S0263574719000560 Published online by Cambridge University Press

CWave, as a fast method for equidistant wave propagation on a grid, can potentially be adapted for single-pair (point-to-point) path planning, and can be used to efficiently solve other geometric problems on a grid, such as painting out circle sectors, segmenting the grid along equidistant, or geodesic lines (extremals) with respect to a given source.

7. Conclusion

In this work, we presented a high-performance algorithm for single-source any-angle path planning on 2D grids. The key idea of the algorithm is to abandon the graph model and operate directly on the grid geometry using discrete geometric primitives (based on Bresenham's circle and line algorithms), instead of individual vertices, to represent the wave front. A detailed analysis of the discrete geometric primitives is presented.

In its most basic form, CWave requires only integer addition and bit shifting to operate. This, however, results in an accumulative error of 0.5 at every turning point. And while such distance error can be ignored in many practical applications, we presented a modified version that uses a minimal number of floating-point operations (only to calculate exact distances between related corner points) that are able to find optimal paths (without accumulative errors). The absence of accumulative errors is demonstrated mathematically and verified by two experimental methods.

The speed of CWave is experimentally compared to the speed of other any-angle path planning algorithms, such as Theta*, Lazy Theta*, Field A*, ANYA, Block A*, and A* modified for single-source path planning. On all three test maps, CWave performed faster than other algorithms (several times faster on the target robot navigation map). A multithreaded implementation of CWave has been described and tested. It demonstrated a noticeable performance improvement (almost three times on some maps), but with N threads it was far from being N-times faster. Some drawbacks of CWave have been identified including the high complexity of the implementation. Multiple directions for possible improvements and for further research have been outlined.

Acknowledgments

This paper is a significantly extended version of ref. [33]. As compared to the conference paper, here we provide all mathematical proofs, pseudo-code for the algorithm, present a multithreaded implementation of CWave, demonstrate additional performance gains achieved with parallelization, and discuss many nuances of the development and usage of the algorithm that were left out in ref. [33] due to page limitations. This material is based upon work supported by the National Science Foundation under Grant No. 1135854. We would like to thank Tansel Uras, who kindly helped us to adapt their test framework[17] for single-source planning.

References

1. D. A. Sinyukov, R. Li, N. W. Otero, R. Gao and T. Padir, "Augmenting a voice and facial expression control of a robotic wheelchair with assistive navigation," *2014 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, San Diego, CA, USA (2014) pp. 1088–1094.
2. D. A. Sinyukov, "Semi-autonomous robotic wheelchair controlled with low throughput human-machine interfaces," *Ph.D. Dissertation*. https://web.wpi.edu/Pubs/ETD/Available/etd-050117-140934/unrestricted/sinyukov-phd-dissertation.pdf.
3. W. van Toll, A. F. Cook and R. Geraerts, "Navigation meshes for realistic multi-layered environments," *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2011) pp. 3526–3532.
4. A. Nash, "Any-Angle Path Planning," *Ph.D. Dissertation* (University of Southern California, 2012). Available: http://gradworks.umi.com/35/42/3542296.html
5. G. K. Kraetzschmar, G. P. Gassull, K. Uhl, G. Pags and G. K. Uhl, "Probabilistic quadtrees for variable-resolution mapping of large environments," **In**: *Proceedings of the 5th IFAC/EURON symposium on intelligent autonomous vehicles*, vol. 37 (2004) pp. 675–680.
6. G. Grisetti, C. Stachniss and W. Burgard, "Improved techniques for grid mapping with Rao-Blackwellized particle filters," *IEEE Trans. Rob.* **23**(1), 34–46 (2007).
7. E. Marder-Eppstein, E. Berger, T. Foote, B. P. Gerkey and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," *2010 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, USA (2010) pp. 300–307.
8. S. Rabin, "A* speed optimizations," **In**: *Game Programming Gems*, vol. 1 (Charles River Media, Boston, MA, USA, 2000) pp. 272–287.
9. A. Nash and S. Koenig, "Any-angle path planning," *AI Mag.* **34**(4), 85–107 (2013).
10. J. Carsten, A. Rankin, D. Ferguson and A. Stentz, "Global planning on the mars exploration rovers: Software integration and surface testing," *J. Field Rob.* **26**(4), 337–357 (2009).

11. C. Thorpe and L. Matthies, "Path relaxation: Path planning for a mobile robot" *OCEANS* (1984) pp. 576–581.
12. A. Nash, K. Daniel, S. Koenig and A. Felner, "Theta*: Any-angle path planning on grids," **In:** *Proceedings of the 22nd National Conference on Artificial Intelligence*, vol. 2 (AAAI Press, Vancouver, Canada, 2007) pp. 1177–1183.
13. A. Nash, S. Koenig and C. Tovey, "Lazy Theta*: Any-angle path planning and path length analysis in 3D," **In:** *Proceedings of the 5th IFAC/EURON symposium on intelligent autonomous vehicles*, Atlanta, GA, USA (2010).
14. S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Trans. Rob.* **21**(3), 354–363 (2005).
15. T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Commun. ACM* **22**(10), 560–570 (1979).
16. K. Daniel, A. Nash, S. Koenig and A. Felner, "Theta*: Any-angle path planning on grids," *J. Artif. Intell. Res.* **39**, 533–579 (2010).
17. T. Uras and S. Koenig, "An empirical comparison of any-angle path-planning algorithms," *Eighth Annual Symposium on Combinatorial Search*, IL, USA (2015).
18. T. Uras, S. Koenig and C. Hernández, "Subgoal graphs for optimal pathfinding in eight-neighbor grids," *ICAPS*, Rome, Italy (2013).
19. T. Uras, *ICAPS 2015: Speeding-up any-angle path-planning on grids* (2010). https://youtu.be/DduMITmi5Ek.
20. J. A. Sethian, "A fast marching level set method for monotonically advancing fronts," *Proc. Natl. Acad. Sci.* **93**(4), 1591–1595 (1996). pMID: 11607632.
21. C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.* **EC-10**(3), 346–365 (1961).
22. J. S. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The discrete geodesic problem," *SIAM Journal on Computing* **16**(4), 647–668 (1987).
23. J. S. B. Mitchell, "Shortest paths among obstacles in the plane," **In:** *Proceedings of the Ninth Annual Symposium on Computational Geometry*, Ser. SCG'93 (ACM, 1993) pp. 308–317. http://doi.acm.org/10.1145/160985.161156.
24. L. D. Elsgolc, *Calculus of Variations* (Dover Publications, Mineola, NY, USA, 2007).
25. J. Bresenham, "A linear algorithm for incremental digital display of circular arcs," *Commun. ACM* **20**(2), 100–106 (1977).
26. J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Syst. J.* **4**(1), 25–30 (1965).
27. D. Harabor and A. Grastien, "An optimal any-angle pathfinding algorithm," **In:** *Proceedings of the Twenty-Third International Conference on International Conference on Automated Planning and Scheduling*, Ser. ICAPS'13, Rome, Italy (AAAI Press, 2013) pp. 308–311. http://dl.acm.org/citation.cfm?id=3038718.3038758.
28. D. Harabor, A. Grastien, D. Öz and V. Aksakalli, "Optimal any-angle pathfinding in practice," *J. Artif. Int. Res.* **56**(1), 89–118 (2016).
29. N. Sturtevant, "Benchmarks for grid-based pathfinding," *IEEE Trans. Comput. Intell. AI Games* **4**(2), 144–148 (2012). http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.
30. A. Williams, *C++ concurrency in action: Practical multithreading*. (Manning, NY, USA, 2012) oCLC: ocn320189325.
31. M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Ng, "ROS: An open-source robot operating system," *ICRA Workshop on Open Source Software*, vol. 3 (2009).
32. D. Sinyukov and T. Padir, "A novel shared position control method for robot navigation via low throughput human-machine interfaces," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2018)*, Madrid, Spain (2018).
33. D. A. Sinyukov and T. Padir, "CWave: High-performance single-source any-angle path planning on a grid," *2017 IEEE International Conference on Robotics and Automation (ICRA)*, Singapore (2017).