Robotica (2019) volume 37, pp. 641–655.
doi:10.1017/S0263574718001236

RimJump: Edge-based Shortest Path Planning for a 2D Map

Zhuo Yao, Weimin Zhang*, Yongliang Shi, Mingzhu Li, Zhenshuo Liang, Fangxing Li and Qiang Huang

School of Mechatronics, Beijing Institute of Technology, Beijing Advanced Innovation Center for Intelligent Robots and Systems, Key Laboratory of Biomimetic Robots and Systems (Beijing Institute of Technology), Ministry of Education, 100081 Beijing, China
E-mails: joe_yao@foxmail.com; ylshi@bit.edu.cn, limingzhu@bit.edu.cn, liangzs_bit@foxmail.com, wonk2000@bit.edu.cn, qhuang@bit.edu.cn

(Accepted October 24, 2018. First published online: November 29, 2018)

SUMMARY
Path planning under 2D map is a key issue in robot applications. However, most related algorithms rely on point-by-point traversal. This causes them usually cannot find the strict shortest path, and their time cost increases dramatically as the map scale increases. So we proposed RimJump to solve the above problem, and it is a new path planning method that generates the strict shortest path for a 2D map. RimJump selects points on the edge of barriers to form the strict shortest path. Simulation and experimentation prove that RimJump meets the expected requirements.

KEYWORDS: Path planning; Shortest path; RimJump.

1. Introduction

Path planning, which refers to providing a path for navigating a movable object toward a desired location from a starting position,[1] has been widely applied in the robotics community[2] and other real-world application fields,[3] such as window cleaning,[4] exploration of Mars, trajectory planning for robot arms,[5] and video games.[6]

In recent decades, various path planning methods have been developed. A classic path planning algorithm based on a grid map is represented by Dijkstra. Methods based on Dijkstra use fewer computing resources, are easy to implement, and have the ability to represent irregular obstacles, which results in a high level of convenience for the implementation of path planning. However, they are unable to determine the shortest path.[7] These algorithms are based on nearby area searching; hence, when the map scale increases, their time cost increase sharply.

We believe that there is a method that can determine the shortest path based on a grid map and its time cost is determined by the complexity of the obstacles rather than the map scale. In daily life, we can quickly determine the shortest path in a simple environment, such as one that only contains a barrier; however, it becomes difficult to determine the shortest path when the barrier becomes increasingly complex.

Thus, we present RimJump, which is a strict shortest global path planning method that uses barrier edge information. Global planning means find a collision-free path for a moving object among stationary, completely known obstacles. And RimJump's time cost is mainly determined by complexity of the obstacles rather than the map scale.

The methodology of RimJump is different from traditional path planning for a grid map. As mentioned in ref. [8], optimal paths must be straight in homogeneous regions. Thus, essentially, a path can be decomposed into a series of line segments, where the endpoints of these line segments are the points at which the direction of the path changes. In the following, we refer to such a segment

* Corresponding author. E-mail: zhwm@bit.edu.cn

as a path unit. By observing nature, we know that the inflection point of the shortest collision-free path must be on the edge of a barrier. Thus, if we can determine the correct inflection point of the path from the edge, we can obtain the shortest path. RimJump determines the correct inflection point from the tangency point on edges. Although RimJump cannot determine the shortest path directly, it obtains a path set that contains the shortest path, and determines the shortest path from the set.

The time cost of RimJump is mainly determined by the complexity of the obstacles and insensitive to the scale of the map. Additionally, it is robust under a variety of complex maps. The result consists of tangents; thus, the path is smoother than that generated by traditional methods.

However, RimJump algorithm still has some drawbacks: it can only be used for static global planning, and cannot be used in map with multiple cost values (the algorithm based on point-by-point traversal does not have this problem).

The remaining part of this article is organized as follows: in Section 2, we introduce some related works. In Section 3, we present all the details of RimJump. In Section 4, we present the results of RimJump, Dijkstra, A*, potential field, ant path planning, and RRT (Rapidly exploring Random Tree) to prove advantages of RimJump. Finally, we present conclusions and future work in Section 5.

2. Related Work

Path planning has been widely studied by robot communities and other researchers, and many methods have been proposed to address practical problems.

The Dijkstra algorithm is a classic graph search algorithm and a typical single-source shortest path algorithm, which determines the shortest path from one node to all the other nodes. It extends outward from the start until it reaches the target. Many path planning algorithms based on Dijkstra have appeared. Likhachev proposed A*.[9] Compared with the Dijkstra algorithm, it adds an estimate of the future cost. In some cases, A* has a speed advantage over the Dijkstra algorithm. RimJump is faster than Dijkstra because it does not need point-to-point traversal when the map scale is big or the obstacle is simple.

Stentz proposed D* (Dynamic A*, Stentz et al.[10]). D* is very effective for planning in a dynamic environment. After D*, Anthony Stentz proposed the focused D* based on D*, which is an extension of D* that focuses on reducing the total time required for initial path calculation and replanning.[11]

Nash, Daniel, and Koenig proposed Theta*.[12] After A* generates the path, Theta* detects whether there is any obstacle between two non-contiguous points on the path. If not, the middle point is deleted and the two points are connected directly. Based on Theta*, Nash, Koenig, and Tovey proposed Lazy Theta*.[12] The core concept of Lazy Theta* is to delay the line-of-sight check until the node is opened. This avoids line-of-sight detection on many points that are not the shortest path. As a result, Lazy Theta* reduces the amount of computation compared with Theta*.

In recent decades, there have been path planning algorithms that have not been based on Dijkstra. Sampling-based path planning algorithms[13–16] generally do not plan on grid maps with minimum grid resolution directly. They use a random density of particles scattered on the map to abstract the actual map to assist planning. Sampling-based path planning has proven to be an effective framework suitable for a large class of problems in domains such as robotics, manufacturing, computer animation, and computational biology.[3,17] These techniques handle complex problems in high-dimensional spaces but usually operate in a binary world aiming to find out collision-free solutions rather than the optimal path. The ant colony algorithm[18–23] simulates the behavior of ants in nature. As mentioned in ref. [23], when searching for food, biological ants exhibit complex social behavior based on the hormones they deposited (called pheromones). Pheromones attract other ants and outline a path to the food source that other ants can follow. As more ants walk along the path, more pheromone is laid, and the chance that more ants will take the path increases. The shortest path to the food builds up the most pheromones because more ants can travel it in less amount of time. So it has strong universality and robustness, and is suitable for searching paths on graphs; however, it is unable to determine the optimal result.

The genetic algorithm[24–31] is based on the principle of survival of the fittest and generates increasingly better approximation solutions from generation to generation. The optimal individuals in the final generation population can be decoded as the approximate optimal solution to the problem. Genetics methods can overcome many problems encountered by traditional search techniques such as the gradient-based methods. This algorithm allows four-neighbor movements, so that path- planning can adapt with complicated search spaces with low complexities.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

The genetic algorithm has strong global search capabilities and is suitable for solving discrete problems, particularly when the crossover probability is large. It can produce a large number of new individuals and improve the global search range. However, genetic algorithm cannot always determine the optimal result too.

RimJump has an idea similar to genetic algorithms. We can treat an iteration in RimJump as generating the next generation of path from the previous one. RimJump deletes path in each iteration based on the principle of survival of the fittest too. The last individual (path) is the optimal path.

Artificial potential fields[32] is a simple and effective real-time obstacle avoidance approach. There are two kinds of artificial potential fields: (1) the manipulator moves in a field of forces, the position to be reached is an attractive pole far the end effector, and obstacles are repulsive surfaces for the manipulator parts; (2) the potential field determined by a certain gradient from the starting point to the target.[33] However, artificial potential field approach has a major problem which is that a robot is trapped at a local minimum before reaching it's goal.

Several approaches are proposed to solve this problem, including simulated annealing[34,35] and virtual obstacle.[36]

The simulated annealing algorithm is simulating the anneal process of metal. When local minimum has occurred, with the simulated annealing approach, a new solution for the candidate of next position is chosen randomly from a set of neighbors of the current solution. The new solution is accepted if the new position has lower potential energy, or else the algorithm proceeds to the next step where the temperature is decreased by cooling rate. This is repeated until a small value near zero is reached or escape from the local minimum has occurred.

The virtual obstacle is located around local minimum point to repel the robot from the point. This technique is useful for the local path planning in unknown environments.

Control Adjoining Cell Mapping and Reinforcement Learning (CACM-RL) path planning[37,38] aims to integrate SLAM into the path planning based on CACM-RL to give a total autonomy to mobile vehicles. Cell mapping means partitions a continuous state space into a finite number of disjoint cells. For path planning, only transitions between adjoining cells are allowed. Reinforcement learning methods only require a scalar reward (or punishment) to learn to map situations (states) in actions. The objective is to find the path that maximizes the accumulated reward in each grid of path. Thus, CACM-RL can get an almost optimal path.

The visibility graph[39–41] views obstacles in configuration space as polygons and then constructs a graph using the start position, goal position, and vertices of polygons. Then the path is obtained using graph search methods, such as the Dijkstra algorithm.

The visibility graph has some similarities with RimJump. They both do not need point-to-point traversal and have the potential to get the optimal path. However, visibility graph needs to reconstruct the visibility graph when the start and the target changes. And it is an algorithm for a polyhedral object moving among known polyhedral objects. If the edge of the obstacle is a smooth curve, as shown in Fig. 2, it is difficult to construct the visibility graph.[42–44]

3. Methodology

In this section, we introduce the methodology and optimization of RimJump.

First, we clarify the requirements of the planning results: (1) the path connects the starting point with the target; (2) the path is collision-free, that is, it does not cross any edge of the obstacle; and (3) the path is the shortest in the set of paths that meet the first two conditions.

Then we discuss how to determine the path that meets all the aforementioned conditions. For a simple barrier, such as that shown in Fig. 1(a), we can easily obtain two possible shortest paths, which are shown in Fig. 1(b), using human observation. When a map becomes increasingly complex, as shown in Fig. 2, it is difficult for a human to determine all possible optimal paths.

From these results, we can see that the inflection point of the shortest path must be on the edge of the barrier. Thus, if we determine the correct inflection point of the path from the edge, then we obtain the shortest path. RimJump determines the correct inflection point from the tangency points on edges. As there is more than one inflection point in a path, we need to select edge points more than once.

RimJump cannot determine the shortest path directly; thus, we obtain a path set that contains the shortest path and determine the shortest path from the set.

For an unfinished path, selecting an inflection point is equivalent to adding a new path unit. In the following, we refer to adding a new path unit as jump. A path from the starting point after several jumps reaches the target. The essence of jump is to determine a tangent from a point to an edge. The point is the last point of an unfinished path and the edge is the edge of the barrier to which the path jumped.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

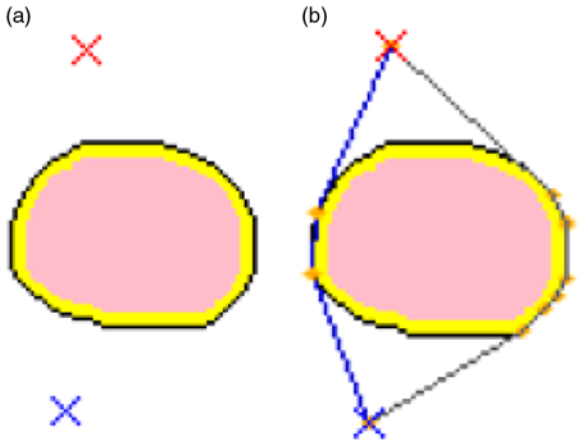

Fig. 1. (a) Simple barrier and (b) path planning for it using human observation. White represents free points, colored patches represent obstacles, and gray curves represent the path. A red 'x' represents the target and a blue 'x' represents the starting point. Yellow points represent the inflection points of the path.

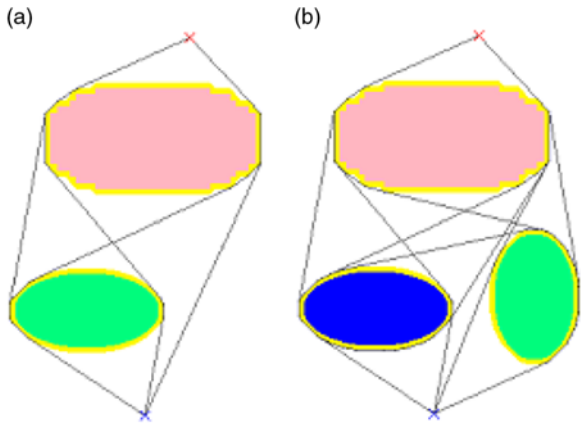

Fig. 2. Possible optimal paths for a map that contains (a) two barriers and (b) three barriers. When barriers become increasingly complex, the total number of possible optimal paths increases sharply, which makes it difficult for humans to determine all of them.

Next, we introduce the details of RimJump, including edge processing, jump, initialization, and iteration. Their relationship is shown in Algorithm: RimJump.

01: **Algorithm**: ***RimJump***
02: **Input**: M, MAX_ITER
03: ORIGIN, TARGET, SAFE_RADIUS,
04: MIN_WIDTH_OF_SUB_EDGE
05: **Output**: *shortestFinishedPath*
06: *subEdges* = ***EdgeProcessing***(M, SAFE_RADIUS, MIN_WIDTH_OF_SUB_EDGE)
07: *initPaths* = ***Initialization***(M, *subRims*, ORIGIN, TARGET)
08: *shortestFinishedPath* = ***Iteration***(M, *initPaths*, *subEdges*, TARGET, MAX_ITER)
09 : **return** *shortestFinishedPath*

3.1. Edge processing

In this subsection, we mainly focus on the treatment of the barrier's edge, including inflation, dividing the barrier, edge detection, edge point sorting, and edge segmentation. It is noteworthy that each map only requires edge processing once, even for different starting and ending positions.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

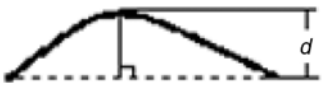

Fig. 3. Solid curve represents a sub-edge and *d* refers the width of the sub-edge.

Fig. 4. The barrier contains both concave and convex sub-edges. If we consider the barrier as an edge rather than divide it into sub-edges, as shown Fig. 5, then we cannot reach the target inside the barrier using a tangent to the entire edge from outside.

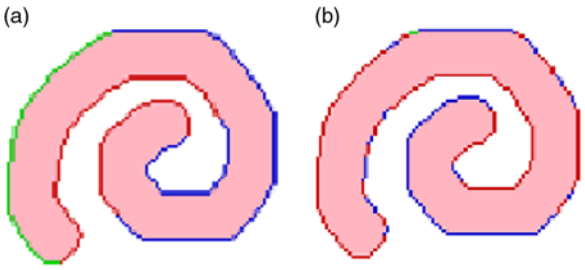

Fig. 5. If we consider the width of sub-edge during edge segmentation, result will be as shown in (a); if not, these sub-edges are too fragmented, as shown in (b).

First, there are two types of cost value, free and obstacle, for each point in the map. If a point is on the edge of a barrier, then it is an edge point.

Given that robots have a certain volume, RimJump expands the barrier outwardly using the inflation radius to avoid robots hitting obstacles.

While the inflation (*L07 and L08*) is processing, we find out all obstacle points in map (*L09 and L10*). Then we divide the obstacles according to the continuity of the obstacle (*L11–L14*). After this step, we know which obstacle the barrier point belongs to.

Dividing the barrier is preparation for the edge detection. For an obstacle, all the points that achieve the condition of "the four points closest to it contain both a barrier point and free point" are selected as edge points, as shown in *L15–L22*.

It is noteworthy that, after edge detection, the order of the points in the container is irregular; that is, adjacent points in the container (container = se in *L16*) are not adjacent in the map.

In preparation for edge segmentation, we need to sort all the edge points and make adjacent points in the container also adjacent in the map (*L23 and L24*).

Then the edge is split into several simple concave and convex edges *(L25–L35*). In the following, we refer to such a simple concave or convex edge as a sub-edge. In the *L32*, width of sub-edge refers the max distance between points on the edge and the line that connects the start and end points of the sub-edge as shown in Fig. 3.

The edge segmentation aims to make RimJump plan successfully when the target is in the depression of a concave edge, as shown in Fig. 4.

We introduce the concept "width of sub-edge" to prevent edge segmentation from being too fragmented, as shown in Fig. 5(b), which would result in extra computational cost. Min width of sub-edge is determined by the scale of the map and the minimum obstacle scale.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

```
01: Function: EdgeProcessing
02: Input: M, SAFE_RADIUS, MIN_WIDTH_OF_SUB_EDGE
03: Output: SUB_EDGES
03:     B = ∅, SB = ∅
04:     SE = ∅, SSE = ∅
05:     SUB_E = ∅
06:     foreach point in M
07:         if DistanceBetween(point, NearstObstaclePointTo(point)) < SAFE_RADIUS then
08:             point.state = OBSTACLE
09:         if point.state = OBSTACLE then
10:             add point to B
11:     for points belongToTheSameBarrier in B
12:         sb = ∅
13:         add points to sb
14:         add sb to SB
15:     foreach barrier in SB
16:         se = ∅
17:         foreach point in barrier
18:             foreach neighbor of point
19:                 if neightbor.state = FREE then
20:                 add point to se
21:                 break
22:       add se to SE
23:     sort all separeteEdge in SE
24:     SSE = SE
25:     foreach sortedSeparateEdge in SSE
26:         tempEdge = ∅
27:         for i = 1 to sizeof(sortedSeparateEdge)
28:           if sizeof(tempEdge) < 2 then
29:               add sortedSeparateEdge[i] to tempEdge
30:               continue
31:           if line(tempEdge. start, sortedSeparateEdge[i]) cross temEdge and
32:           widthOf(tempEdge) > MIN_WIDTH_OF_SUB_EDGE
33:               add tempEdge to SUB_EDGES
30:               tempEdge = ∅
34:           else
35:               add sortedSeparateEdge[i] to tempEdge
36:     return SUB_EDGES
```

3.2. Jump

Jump is the essence of RimJump. Jump means select an inflection point on a sub-edge and add it to an unfinished path. This process is called "jump to the sub-edge." In *L04,* function line(p1, p2) means construct a line that connects p1 and p2.

We only accept tangents that meet three requirements (*L07*, *L08*, and *L09*).

Because of conditions 2 and 3, the available range of the new path unit's direction is only 90°, as shown in Fig. 6.

These conditions can greatly reduce the generation of useless path units, thereby improving the efficiency of RimJump.

```
01: Function: Jump
02: Input: M, PATH, SUB_EDGE, TARGET
03: Output: newPaths
03:     newPaths = ∅
04:     if line(PATH. end, TARGET) CrossAnyBarrier then
```

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

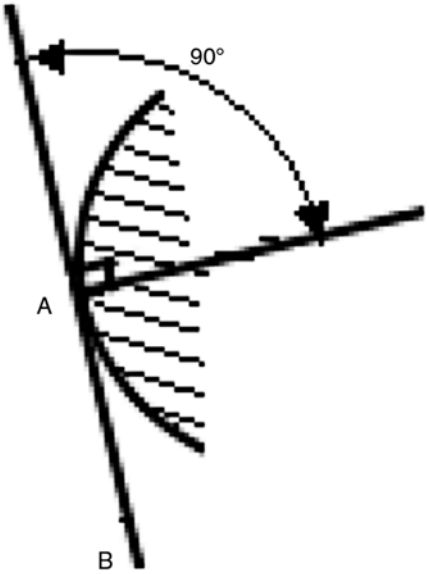

Fig. 6. Shaded area is part of the obstacle, BA is the last path unit of the current path, and 90° is the available range of orientation of the new path unit.

```
05:       tangents = SearchTangents(PATH. end, SUB_EDGE)
06:       foreach tangent in tangents
07:          if tangent NotCrossAnyBarrier and
08:          angleBetween(tangent, PATH.lastPathUnit) ≤ 90° and
09:          tangent IsBiasedToObstacleThan(PATH.lastPathUnit) then
10:             tempPath = PATH
11:             add tangent. end to tempPath
12:             add tempPath to newPaths
13:    else
14:       add TARGET to PATH
15:       add PATH to newPaths
16:    return newPaths
```

3.3. Initialization

After edge processing, it is time for initialization. After initialization, we get the initial paths for iteration. In Fig. 7, we show how to select sub-edge for jump. In Fig. 8, we show how to search tangents. In *L09*, function Path(p1, p2) means construct a path that only contains two points(p1, p2).

```
01: Function: Intialization
02: Input: M, SUB_EDGES, ORIGIN, TARGET
03: Output: initialPaths
03:    selectedEdges = SearchSubEdgeForJump(SUB_EDGES, ORIGIN, TARGET)
04:    initialPaths = ∅
05:    foreach subRim in selectedEdges
06:       tangents = SearchTangent(ORIGIN, subEdge)
07:       foreach tangent in tangents
08:          if tangent NotCrossAnyBarrier then
09:             add Path(ORIGIN, tangent. end) to intialPaths
10:    return initialPaths
```

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

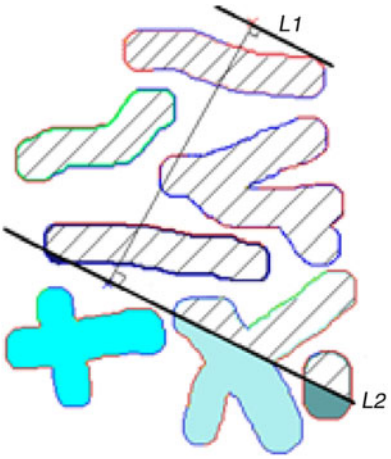

Fig. 7. Approach used to select a barrier between the start point and target: both L1 and L2 are perpendicular to the connection between the last point of current path and target. All barriers contained in the shadowed area are the selected barriers. All sub-edges that belong to the selected barrier are selected sub-edges for jump.

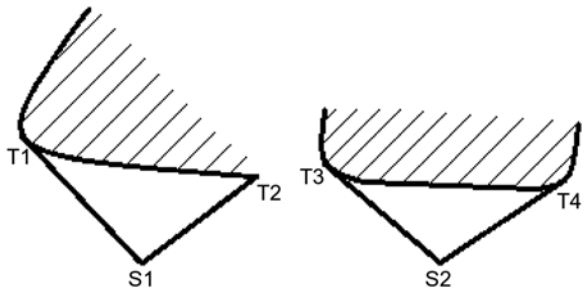

Fig. 8. S1 and S2 represent the ends of an unfinished path; and T1, T2, T3, and T4 represent tangent points. They show how we select a tangent for different scenarios. If there is only a tangent for a sub-edge, then we select the end point of the sub-edge as an inflection point, as T2 shows.

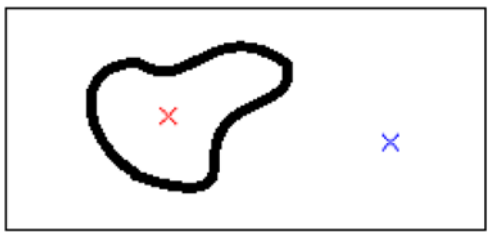

Fig. 9. Although both the start point and target are free points, there is no path to connect them.

3.4. Iteration

Iteration is the core of RimJump.

Because of Line 14 and Line 15, the number of paths that participate in the iteration can be reduced significantly. If a path cannot generate a new path unit, then it is removed.

The number of iterations is limited to prevent the algorithm from becoming stuck in an infinite loop when the target is completely surrounded by obstacles, as shown in Fig. 9.

In the iteration, the number of paths involved in each iteration increases first and then decreases. The turning point often appears after the first completed path appears. Figure 10 shows the relationship between the number of paths and number of iterations for Fig. 11. A typical iteration is shown in Fig. 11.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

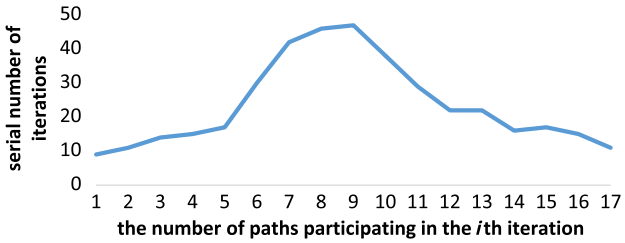

Fig. 10. Relationship between the number of paths and number of iterations.

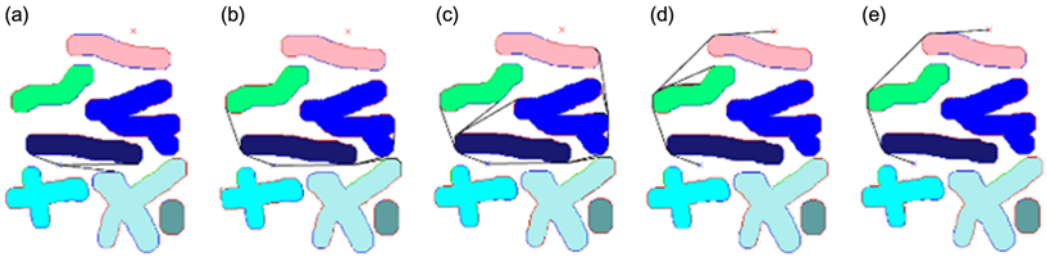

Fig. 11. How paths change during iterations, arranged in chronological order: (a) paths after initialization; (b)–(d) intermediate states during iterations; and (e) the optimal path.

```
01: Function: Iteration
02: Input: M, initialPaths, SUB_EDGES, TARGET, MAX_ITER
03: Output: shortestFinishedPath
03:     sfpl = 0
04:     iterCount = 0
05:     tempPaths = ∅
06:     foreach path in initialPaths
07:         ssefj = SearchSubEdgeForJump(SUB_EDGES, path. end, TARGET)
08:         foreach subEdgeNum in ssefj
09:             add result of Jump(SUB_EDGES[subEdgeNum], initialPath. end, TARGET) to
                    tempPaths
09:         foreach path in temPaths
10:             if path. end = TARGET then
11:                 sfpl = lengthOf(shortestFinishedPath in tempPaths)
12:         if sfpl > 0 then
13:             foreach path in tempPaths
14:                 if lengthOf(path) > sfpl
15:                     remove path from tempPaths
16:         if all path in tempPaths reach TARGET or
17:         iterCount > MAX_ITER
18:                 return shortestFinishedPath in tempPaths
19:         else
20:                 initialPaths = tempPaths
21:                 iterCount = iterCount + 1
22:                 goto Line 05
```

4. Results

In this section, we evaluate the performance of our method and compare it with other related path planning techniques in simulation and real data.

4.1. Comparison with other algorithms

We ran RimJump, Dijkstra, potential field, RRT path planning, ant path planning, and Theta* for several typical maps with the same scale, as shown in Fig. 12.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

Table I. Path length (pixel) of RimJump and other algorithms. 'R' denotes RimJump, 'D' denotes Dijkstra, and 'T*' denotes Theta*.

	A	B	C	D
R	**352.9**	**483.4**	**315.5**	**298.9**
D	363.9	499.5	333.4	315.2
A*	400.0	512.3	367.2	352.9
PF	401.4	Min local	Min local	313.3
Ant	12004.1	14221.1	3163.4	1967.2
RRT	456.3	712.7	496.2	402.2
T*	351.9	483.7	315.1	299.3

Bold values means the shortest path length under different maps.

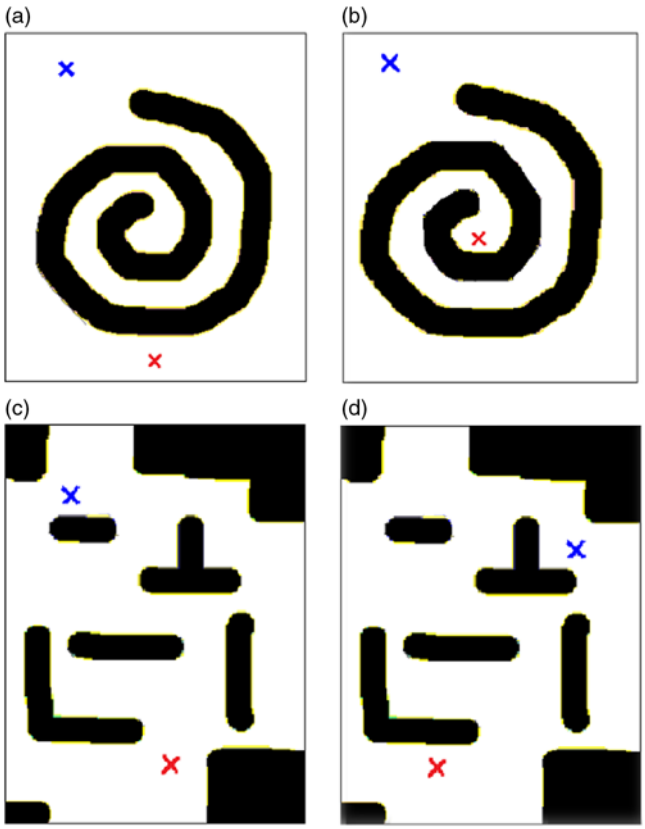

Fig. 12. Maps for comparison. 'x' refers the start and the target.

For Dijkstra and A*, each point in the map corresponds to one node in the graph, and the cost value of the obstacle is set to be gigantic, and the free area is 1. In Tables I and II, "min local" means potential field path planning fall in a local minimum and failed to reach the target.

From Tables I and VI, it can be seen that the path generated by RimJump is the shortest and the speed of RimJump is almost the fastest.

4.2. Relationship between time cost and map scale

From the above, only Dijkstra and A* can be compared with RimJump in terms of performance. Theta* can generate the optimal path, but it's too slow. So we compare RimJump with Dijkstra and A* to show RimJump insensitivity to map scale.

We doubled, tripled, and quadrupled the map shown in Fig. 13 to obtain four maps. The coordinates of the start point and target of the path were enlarged as the map zoomed in so that the relative

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

Table II. Time cost of RimJump and other algorithms.

	A	B	C	D
R	1.4 ms	**4.4 ms**	1.75 ms	**0.53 ms**
D	4 ms	5 ms	2 ms	2 ms
A*	**1 ms**	6 ms	**1 ms**	0.7 ms
PF	4.1 ms	Min local	Min local	4.2 ms
Ant	227 s	256 s	22.8 s	17.56 s
RRT	18 s	2.3 s	17 s	3.8 s
T*	26.2 ms	26.8 ms	59.4 ms	50.8 ms

Table III. Time cost (ms) of RimJump, Dijkstra, and A* for four maps with the same content but different scales. 'MS' represents the map size (pixel*pixel).

MS	300*400	600*800	1200*1600	2400*3200
R	7	**15**	**43**	**87**
D	5	20	70	338
A*	**4**	16	60	174

Table IV. Length (pixel) of the paths generated by RimJump, Dijkstra, and A* for four maps with the same content but different scales.

MS	300*400	600*800	1200*1600	2400*3200
R	**327.6**	**656.0**	**1312.9**	**2626.5**
D	346.5	693.5	1388.0	2774.6
A*	371.7	749.1	1472.3	2983.1

positions of the start point and target, and obstacle remained unchanged. Then we ran RimJump, Dijkstra, and A*, and the planning results and time cost were recorded, as shown in Tables III and IV.

In Table III, we observe that when the scale of the map increased, the time cost of RimJump increased much slower than that of Dijkstra and A*; that is, compared with Dijkstra and A*, RimJump was insensitive to the map scale increasing.

This proves that RimJump was insensitive to the scale of the map. This means that when the map size increased, RimJump was faster than traditional point-to-point traversal path planning methods.

This phenomenon can be explained theoretically. When the length and width of a map are doubled, the acreage is quadrupled and the perimeter is doubled.

Thus, the search area of Dijkstra and A* is quadrupled, the input of RimJump is doubled, and then the time cost of RimJump is doubled whereas that for the other two methods is quadrupled. The simulation results are in line with the above inference, approximately.

4.3. Relationship between time cost and obstacle complexity

From the above, only Dijkstra and A* can be compared with RimJump in terms of performance. So we compare RimJump with them to show RimJump insensitivity to map scale.

We ran RimJump for four maps with the same scale but different content, as shown in Fig. 15. Because it is edge-based, as shown in Table V, there is a strong positive correlation between the time cost of RimJump and obstacle complexity.

This phenomenon can be explained theoretically.

The more complex the map, the more the sub-edges, thus, the time cost increases synchronously; that is, the simpler the map, the faster the RimJump. By contrast, Dijkstra and A* took a great deal of time for the simple map shown in Fig. 15(a).

4.4. Experiments on real map

We transplant RimJump to ROS and then let the robot move according to the path we got in the real environment. We did four experiments in the same scene, and the results are shown in the Fig. 14.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

Table V. Time cost (ms) of RimJump, Dijkstra, and A* corresponding to Fig. 15.

	A	B	C	D
R	**1**	**4**	**15**	45
D	89	85	76	72
A*	32	36	40	**44**

Table VI. Length (pixel) of paths generated by RimJump, Dijkstra, and A* corresponding to Fig. 16.

	A	B	C	D
R	**1607.8**	**1670.9**	**1670.9**	**1700.6**
D	1760.0	1760.0	1760.0	1760.2
A*	1932.1	1931.0	1900.0	1919.94

Fig. 13. Original map with start and end positions: blue ‘x’ represents the start point and red ‘x’ represents the end point.

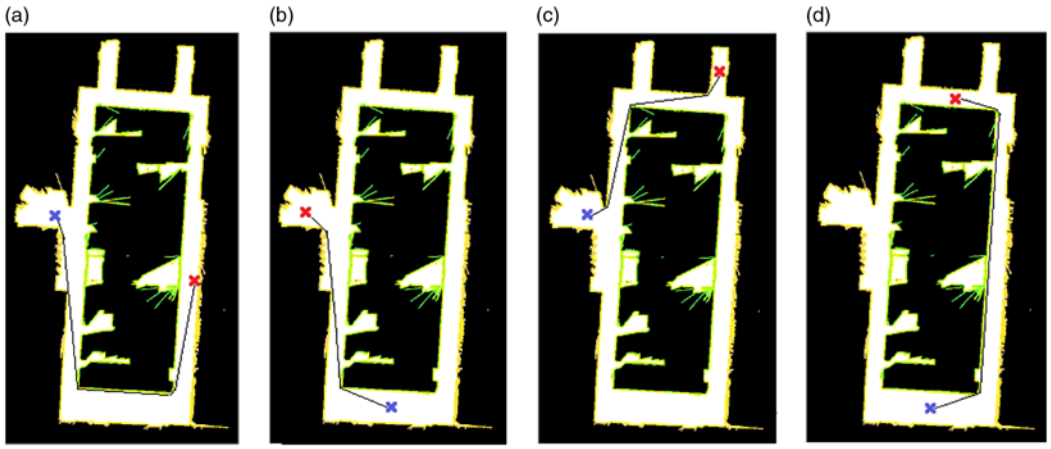

Fig. 14. The map is part of our laboratory. The path is provided by RimJump.

As shown in Fig. 14, RimJump runs fine in the real environment.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

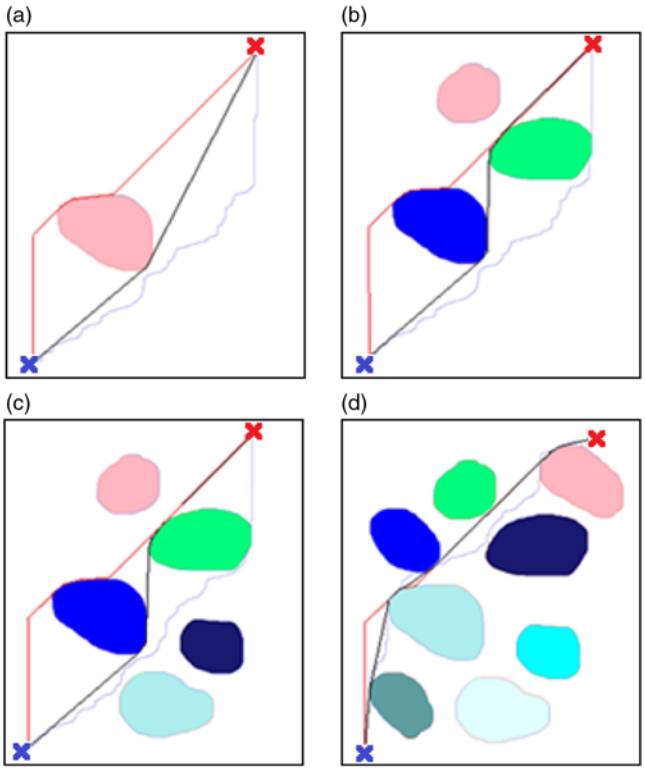

Fig. 15. (a)–(d) Path generated by three methods for four same-scale maps. From (a) to (b), the complexity of the barrier gradually increases. Black curves represent the path generated by RimJump, red curves represent Dijkstra, and blue curves represent A*.

5. Conclusions and Future Work

5.1. Conclusions

According to the principle of RimJump, the inflection point of the shortest collision -free path must be on the edge of the barrier if we are to obtain the strict shortest collision -free path, as shown in Fig. 15.

Because it is edge-based, the speed of RimJump is closely related to the complexity of the obstacles and is insensitive to map scale, whereas traditional grid-based path planning algorithms are often closely related to map scale. Therefore, RimJump is faster than traditional grid-based methods when the map is large or simple. We verified this using a simulation. Insensitivity to the scale of the map allows us to increase the resolution of the map and obtain more accurate environmental information for more accurate planning.

The path generated by RimJump consists of path units, whereas the path generated by the general grid method (like Dijkstra, potential field, ant path planning, and RRT) is often composed of a large number of adjacent points; that is, the path obtained by RimJump takes up less storage space.

It is noteworthy that RimJump is a global planning method, which means, it needs information about environment before planning. So it cannot be applied in an unknown environment.

5.2. Future work

At every iteration, every route from the previous iteration jumps to all edges that lie between the end point of the route and the target. The application of sub-edge sequence information is expected to significantly reduce the number of edges involved in the iteration. By reducing unnecessary jumps, the performance of RimJump can be improved greatly.

For RimJump, the cost value of each map grid must be zero or one; it is not suitable for maps that have various cost values. This problem needs to be solved.

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press

After edge processing, whether the target has been surrounded by obstacles (as shown in Fig. 14) should be checked to avoid wasting computing resources in the case in which the start point and target are not within obstacles but there is no collision-free path.

Acknowledgments
We thank Dr. Hua Zhang, Dr. Gao Huang, Dr. Libo Meng, Dr. Xiongjun Liu, and Haichao Wang. The authors engaged in useful discussions with them and they provided many inspiring suggestions. The first author also thanks Yanan Shi for her support and encouragement during the design of RimJump. We thank Maxine Garcia, PhD, from Liwen Bianji, Edanz Group China (www.liwenbianji.cn/ac) for editing the English text of a draft of this manuscript.

Funding
The work was financially supported by the Pre-Research Project under Grant 41412040101.

Supplementary material
To view supplementary material for this article, please visit https://doi.org/10.1017/S0263574718001236.

References
1. W. Gong, X. Xie and Y.-J. Liu, "Human experience-inspired path planning for robots," *Int. J. Adv. Robot. Syst.* **15** (2018). doi:10.1177/1729881418757046.
2. G. A. S. Pereira, L. C. A. Pimenta, L. Chaimowicz, A. R. Fonseca, D. S. C. de Almeida, L. D. Q. Corrêa, R. C. Mesquita, L. Chaimowicz, D. S. C. de Almeida and M. F. M. Campos, "Robot navigation in multi-terrain outdoor environments," *Int. J. Robot. Res.* **28**(6), 685–700 (2009).
3. S. Lavalle, *Planning Algorithms* (Cambridge University Press, New York, 2006).
4. M. Farsi, K. Ratcliff, P. J. Johnson, C. R. Allen, K. Z. Karam and R. Pawson, "Robot control system for window cleaning," *Autom. Robot. Constr. XI* **1**, 617–623 (1994).
5. G. Sahar and J. M. Hollerbach, "Planning a minimum-time trajectories for robot arms," *Int. J. Robot. Res.* **5**(3), 90–100 (1984).
6. P. Yap, N. Burch, R. C. Holte and J. Schaeffer, "Any-angle path planning for computer games," *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment,* Stanford, California, USA (AAAI Press 2011) pp. 201–207.
7. A. Nash, S. Koenig and C. A. Tovey, "Lazy Theta*: Any-angle path planning and path length analysis in 3D," *Symposium on Combinatorial Search, SOCS,* Stone Mountain, Atlanta, Georgia, USA*, July DBLP* (2010) pp. 299–307.
8. N. C. Rowe and R. F. Richbourg, "An efficient Snell's law method for optimal-path planning across multiple two-dimensional, irregular, homogeneous-cost regions," *Int. J. Robot. Res.* **9**(6), 48–66 (1990).
9. M. Likhachev, D. Ferguson, G. Gordon, A. Stentz and S. Thrun, "Anytime dynamic A*: An anytime, replanning algorithm," *Fifteenth International Conference on International Conference on Automated Planning and Scheduling,* Monterey, CA, USA (AAAI Press, 2005) pp. 262–271.
10. A. Stentz, "Optimal and efficient path planning for partially-known environments," *Proceedings of the IEEE International Conference on Robotics and Automation,* San Diego, CA, USA, 2002, vol. 4 (IEEE, 1994) pp. 3310–3317.
11. A. Stentz, "The focussed D* algorithm for real-time replanning," *International Joint Conference on Artificial Intelligence,* Montreal, Quebec, Canada (Morgan Kaufmann Publishers Inc., San Francisco, CA, 1995) pp. 1652–1659.
12. A. Nash, K. Daniel, S. Koenig and A. Felner, "Theta*: Any-angle path planning on grids," *J. Artif. Intell. Res.* **39**(1), 533–579 (2014).
13. L. Jaillet, J. Cortés and T. Siméon, "Sampling-based path planning on configuration-space costmaps," *IEEE Trans. Robot.* **26**(4), 635–646 (2010).
14. S. Thomas, P. Coleman and N. M. Amato, "Reachable distance space: Efficient sampling-based planning for spatially constrained systems," *Int. J. Robot. Res.* **29**(7), 916–934 (2010).
15. S. M. Persson and I. Sharf, "Sampling-based A* algorithm for robot path-planning," *Int. J. Robot. Res.* **33**(13), 1683–1708 (2014).
16. Z. Sun, D. Hsu, T. Jiang, H. Kurniawati and J. H. Reif, "Narrow passage sampling for probabilistic roadmap planning," *IEEE Trans. Robot.* **21**(6), 1105–1115 (2005).
17. H. Choset, G. A. Kantor, S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations* (MIT Press, Cambridge, MA, 2005).
18. L. M. Gambardella and M. Dorigo, "Ant-Q: A reinforcement learning approach to the traveling Salesman problem," *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning,* Tahoe City, California, USA, July DBLP (1995) pp. 252–260.

19. G. Liu, Y. Peng and X. Hou, "The ant algorithm for solving robot path planning problem," *International Conference on Information Technology and Applications,* Sydney, NSW, Australia (IEEE Computer Society, 2005) pp. 25–27.
20. Q. B. Zhu and Y. Zhang, "An ant colony algorithm based on grid method for mobile robot path planning," *Robot* **27**(2), 132–136 (2005).
21. N. A. Vien, N. H. Viet, S. G. Lee and T. C. Chung, "Obstacle avoidance path planning for mobile robot based on ant-Q reinforcement learning algorithm," *International Symposium on Neural Networks,* Nanjing, China, vol. 4491 (Springer, Berlin, Heidelberg, 2007) Vol. 4491, pp. 704–713.
22. J. Zhou, G. Dai, D-Q. He, J. Ma and X-Y. Cai, "Swarm intelligence: Ant-based robot path planning," *Fifth International Conference on Information Assurance and Security IEEE Computer Society,* Xi'an, China (2009), pp. 459–463.
23. M. Brand, M. Masuda, N. Wehner and X. H. Yu, "Ant colony optimization algorithm for robot path planning," *International Conference on Computer Design and Applications,* Qinhuangdao, China, vol. 3 (IEEE, 2010) pp. V3-436–V3-440.
24. A. Nearchou and N. A. Aspragathos, "A genetic path planning algorithm for redundant articulated robots," *Robotica* **15**(2), 213–224 (1997).
25. A. C. Nearchou, "Path planning of a mobile robot using genetic heuristics," *Robotica* **16**(5), 575–588 (1998).
26. R. Bohlin and L. E. Kavraki, "Path planning using lazy PRM," *Proceedings of ICRA IEEE International Conference on Robotics and Automation,* San Francisco, CA, USA, vol. 1 (IEEE, 2000) pp. 521–528.
27. J. Tu and S. X. Yang, "Genetic algorithm based path planning for a mobile robot," *IEEE International Conference on Robotics and Automation,* Taipei, Taiwan, vol. 1 (2003) pp. 1221–1226.
28. K. H. Sedighi, K. Ashenayi, T. W. Manikas, R. L. Wainwright and H-M. Tai, "Autonomous local path planning for a mobile robot using a genetic algorithm," *IEEE Congress on Evolutionary Computation,* Portland, OR, USA (2004).
29. J. M. Ahuactzin, E. G. Talbi, P. Bessière and E. Mazer, "Using genetic algorithms for robot motion planning," *Selected Papers from the Workshop on Geometric Reasoning for Perception and Action*, vol. 708 (Springer-Verlag, Berlin, Heidelberg, 2006) pp. 84–93.
30. I. Al-Taharwa, A. Sheta, and M. A. Weshah, "A mobile robot path planning using genetic algorithm in static environment," *J. Comput. Sci.* **4**(4), 341–344 (2008).
31. C.-C. Tsai, H.-C. Huang and C.-K. Chan, "Parallel elite genetic algorithm and its application to global path planning for autonomous robot navigation," *IEEE. T. Ind. Electron.* **58**(10), 4813–4821 (2011).
32. O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *Int. J. Robot. Res.* **5**(1), 90–98 (1986).
33. G. Francis, L. Ott and F. Ramos, *Stochastic functional gradient path planning in occupancy maps*, Preprint arXiv:1705.05987 (2017).
34. Q. Zhu, Y. Yan and Z. Xing, "Robot path planning based on artificial potential field approach with simulated annealing," *International Conference on Intelligent Systems Design and Applications,* Jinan, China (IEEE, 2006), pp. 622–627.
35. M. G. Park and M. C. Lee, "Experimental evaluation of robot path planning by artificial potential field approach with simulated annealing," *Proceedings of the, IEEE SICE Conference,* Osaka, Japan, vol. 4 (2003) pp. 2190–2195.
36. M. C. Lee and M. G. Park, "Artificial potential field based path planning for mobile robots using a virtual obstacle concept," *Proceedings 2003 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2003),* Kobe, Japan, Japan, Vol. 2 (IEEE, 2003) pp. 735–740.
37. T. Arribas, M. Gómez and S. Sánchez, "Optimal motion planning based on CACM-RL using SLAM," **44**(8), 75–80 (2012).
38. D. Meziat, "Optimal motion planning by reinforcement learning in autonomous mobile vehicles," *Robotica* **30**(2), 159–170 (2012).
39. T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Commun. ACM* **22**(10), 560–570 (1979).
40. M. de Berg, *Computational Geometry* (Springer, Berlin, 2013).
41. P. Leven and S. Hutchinson, "Toward real-rime path planning in changing environments," **In**: *Algorithmic and Computational Robotics: New Directions: The Fourth International Workshop on the Algorithmic Foundations of Robotics* (B. R. Donald et al., eds.) (A. K. Peters, Wellesley, MA, 2018) pp. 363–376.
42. M. De Berg, M. Van Kreveld, M. Overmars and O. Schwarzkopf, "Computational geometry: Algorithms and applications," *Math. Gaz.* **19**(3), 333–334 (2008).
43. N. Lyle and P. Tomasz, "Geometry for robot path planning," *Robotica* **25**(6), 691–701 (2007).
44. H. Michael, A. S. Matveev and A. V. Savkin, "Algorithms for collision-free navigation of mobile robots in complex cluttered environments: A survey," *Robotica* **33**(3), 463–497 (2015).

https://doi.org/10.1017/S0263574718001236 Published online by Cambridge University Press