Fast and Optimal Planner for the Discrete Grid-Based Coverage Path-Planning Problem

Jaël Champagne Gareau[0000−0002−1906−4157], Éric Beaudry[0000−0002−4460−0556], and Vladimir Makarenkov[0000−0003−3753−5925]

Université du Québec à Montréal
`champagne_gareau.jael@courrier.uqam.ca`
`{beaudry.eric,makarenkov.vladimir}@uqam.ca`

Abstract. This paper introduces a new algorithm for solving the discrete grid-based coverage path-planning (CPP) problem. This problem consists in finding a path that covers a given region completely. Our algorithm is based on an iterative deepening depth-first search. We introduce two branch-and-bound improvements (Loop detection and Admissible heuristic) to this algorithm. We evaluate the performance of our planner using four types of generated grids. The obtained results show that the proposed branch-and-bound algorithm solves the problem optimally and orders of magnitude faster than traditional optimal CPP planners.

Keywords: Coverage path-planning · Iterative Deepening Depth-First Search · Branch-and-Bound · Heuristic Search · Pruning · Clustering

1 Introduction

Path Planning (PP) is a research area aimed at finding a sequence of actions that allows an agent (e.g., a robot) to move from one state (e.g., position) to another [8] (e.g., finding an optimal path for an electric vehicle [1]). One of the problems studied in PP is the complete Coverage Path-Planning (CPP) problem. Basically, the objective of CPP is to find a complete coverage path of a given region, i.e., a path that covers every area in the region. This problem has many practical applications, including window washer robots, robotic vacuum cleaners, autonomous underwater vehicles (AUVs), mine sweeping robots, search and rescue planning, surveillance drones, and fused deposition modeling (FDM) 3D printers. All of them rely on efficient CPP algorithms to accomplish their task [3,6,7].

There exist many variants of the CPP problem. For example, the environment can be either discrete (e.g., grid-based, graph-based, etc.) or continuous, 2D or 3D, known *a priori* (off-line algorithms), or discovered while covering it (on-line algorithms), etc. Moreover, the coverage can be done by a single agent, or by the cooperation of multiple agents. Some variants also restrict the type of allowed movements or add different kinds of sensors to the agent (proximity sensor, GPS, gyroscopic sensor, etc.). Some variants even consider positional uncertainties and energetic constraints of the agent. In this paper, we focus on the classic variant consisting of a single agent in a 2D discrete grid-based environment with no specific constraints or uncertainties.

The objective of our study is to present an optimal CPP planner that runs orders of magnitude faster than a naive search through the state-space. Our research contributions are as follows:

1. A novel branch-and-bound optimal planner to the grid CPP problem;
2. An informative, admissible, efficient heuristic to the grid CPP problem;
3. Realistic environments for benchmark;

The rest of the paper is structured as follows. Section 2 presents a short overview of existing CPP solving approaches. Section 3 formally introduces the CPP variant we are focusing on. Section 4 and Section 5 present, respectively, our method and the obtained results. Finally, Section 6 describes the main findings of our study and presents some ideas of future research in the area.

2 Related Work

One of the most known algorithm which solves the CPP problem in a grid-based environment is the wavefront algorithm [14]. Given a starting position and a desired arrival (goal) position (which can be the same as the starting position), the wavefront algorithm propagates a *wave* from the goal to the neighboring grid cells (e.g., with a breadth-first search through the state-space). After the propagation, every grid cell is labeled with a number that corresponds to the minimum number of cells an agent must visit to reach the goal from the cell. The algorithm is then simple: the agent always chooses to visit the unvisited neighboring cell with the highest number first, breaking ties arbitrarily. One disadvantage of this strategy is the obligation to specify a goal state. In some applications, the ending location is not important, and not specifying it allows for finding shorter paths. There exists an *on-line* variant of the wavefront algorithm [12] that can be used if the environment is initially unknown. When a CPP algorithm should be applied to a road network (e.g., a street cleaning vehicle that needs to cover every street), a graph representation (instead of a grid) is advantageous. Many algorithms have been proposed to solve the discrete CPP in graph-based representation [13].

For the coverage of continuous regions, one major family of algorithms is the *cellular decomposition methods*. They consist in partitioning complex regions in many simpler, non-overlapping regions, called cells. These simpler regions don't contain obstacles, and are thus easy to cover. The most known algorithm that uses this strategy is the *boustrophedon decomposition* algorithm [4]. Another strategy that can be used for the complete coverage of continuous regions is to discretize the environment and use a discrete planner (e.g., the aforementioned wavefront algorithm).

The above algorithms are relatively fast, but provide no guarantee of optimality. This is expected, since a reduction exists between the CPP problem and the Travelling Salesman Problem, making the CPP problem part of the

NP-Complete class of problems [10]. Thus, to the best of our knowledge, all CPP planners described in the literature either provide approximate solutions, or work only in a specific kind of environment.

For related works on more specific variants of the CPP problem, see the referred surveys [3,6,7].

3 Problem modeling

This paper presents an optimal planner which, since the problem is NP-Complete, must have a worst-case exponential complexity. However, we propose two speed-up methods that, according to our evaluation, provide an orders of magnitude faster planner. We focus on the CPP variant consisting in a 2D grid that needs to be covered by a single agent with no particular goal position. Definitions 1 to 3 describe more formally the environment to cover, the agent and the state-space model considered in our study.

Definition 1. *A 2D* environment *is an $m \times n$ grid represented by a matrix $G = (g_{ij})_{m \times n}$, where $g_{ij} \in \{O, X\}$, and:*

- *O indicates that a cell is accessible and needs to be covered;*
- *X means that the cell is inaccessible (blocked by an obstacle).*

Definition 2. *An* agent *is an entity with a position somewhere on the grid. It can move to neighboring grid cells by using an action a from the set of actions $\mathcal{A} = \{(-1,0), (+1,0), (0,-1), (0,+1)\}$. The effect of each action is as follows.*

If $p = (i,j)$ denotes the agent's current position (i.e., the agent is on the grid cell g_{ij}) and it executes action $a = (a_1, a_2)$, then its new position $\tilde{p}$ is:

$$\tilde{p} = \begin{cases} (i + a_1, j + a_2) & \text{if } g_{i+a_1, j+a_2} = O \\ p & \text{if } g_{i+a_1, j+a_2} = X. \end{cases}$$

Definition 3. *A* state *is a tuple $s = (i_s, j_s, R)$, where:*

- *(i_s, j_s) is the position (row, column) of the agent;*
- *$R = \{(i,j) \mid g_{ij} = O$ and position (i,j) has not yet been explored$\}$.*

We now formally define what we mean by a CPP problem instance, a solution to such an instance, and our optimization criterion in Definitions 4 to 6, and give an example of a problem instance in Fig. 1.

Definition 4. *An* instance *of our CPP problem variant is given by a tuple (G, s_0), where G is an environment, as defined in Definition 1, and $s_0 = (i_0, j_0, R_0)$ is the initial state, where $R_0 = \{(i,j) \mid g_{ij} = O\}$.*

Definition 5. *A* solution *to such an instance (G, s_0) is an ordered list of actions $p = \langle a_1, a_2, \ldots, a_k \rangle$ (also called a plan) that moves the agent through positions:*

$$L = \langle (i_0, j_0), (i_1, j_1), \ldots, (i_k, j_k) \rangle,$$

with $R_0 \subseteq L$ (i.e., the final state is $(i_k, j_k, \emptyset)$).

Definition 6. *Let $\mathcal{P}$ be the set of solutions (plans) of a CPP problem instance. The* objective *is to find an optimal solution $p^\star = \operatorname{argmin}_{p \in \mathcal{P}} |p|$. Namely, $p^\star$ is a minimal ordered list of actions that solves the problem.*

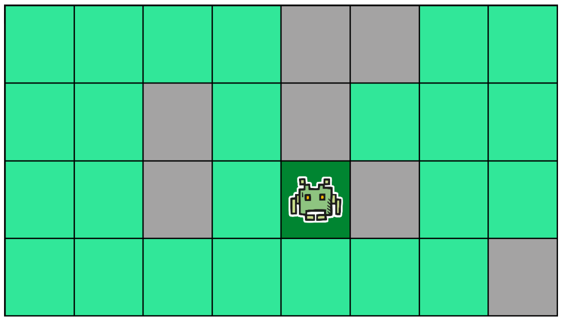


Fig. 1. A CPP instance. The dark green, light green and grey cells represent, respectively, the initial cell, the cells that remain to be covered and the inaccessible cells.

4 Proposed Methods

In this section, we present an optimal CPP planner based on the ID-DFS algorithm, and describe two techniques (loop detection and an admissible heuristic) that preserve the optimality of the obtained solutions, while running orders of magnitude faster than an exhaustive search algorithm.

4.1 Iterative Deepening Depth-First Search (ID-DFS)

We begin by describing a naive planner first, since our branch-and-bound algorithm is based on it, and since no optimal CPP planners are explicitly mentioned in the literature. First, we observe that our problem can be viewed as a search in a graph where every node represents a state in the state-space (it is worth noting that such a graph is exponentially larger than the problem grid). Thus, every standard graph search algorithms can theoretically directly be used, including the well-known depth-first search (DFS) and breadth-first search (BFS) algorithms. However, because of the huge size of the search graph, BFS is impractical. Indeed, BFS needs in the worst-case scenario to store the complete state-space graph in the computer's memory, which is too large even for a small problem size (e.g., a 20×20 grid). Thus, every technique based on BFS, including the Dijkstra and A* algorithms, are non-applicable in practice.

On the other hand, the DFS algorithm can go arbitrarily deep in the search tree even though the solution is close to the root (e.g., it can get stuck by expanding nodes over and over indefinitely, never backtrack and thus never find a solution). A safeguard is to use a variant of DFS, called iterative deepening depth-first search (ID-DFS) [11], which uses DFS but has a depth limit. If a solution is not found within the depth limit k, DFS is carried out again with the depth limit $k + 1$ and continues until a solution is found. ID-DFS ensures that the algorithm never goes deeper than necessary and ensures that the algorithm terminates (if a solution exists). Algorithm 1 presents the details of a CPP planner based on ID-DFS. We recall from the previous section that R is the set of grid cells that remain to be covered (i.e., initially, it is equal to the total set of grid cells to cover). The rest of the pseudocode should be self-explanatory.

Algorithm 1 CPP planner based on ID-DFS

```
 1: global
 2:     $p^\star$: data-structure (eventually) containing the solution
 3:     $s = (i_s, j_s, R)$: current agent position
 4: procedure ID-DFS-PLAN( )
 5:     for $k \leftarrow |R_0|$ to $\infty$ do                          ▷ $k$ is the depth limit
 6:         $found \leftarrow$ ID-DFS-HELPER$(k, 0)$
 7:         if $found$ then return
 8: procedure ID-DFS-HELPER($k$: depth-limit, $d$: current depth) : boolean
 9:     if $k = d$ then return $|R| = 0$          ▷ returns true iff the grid is fully covered
10:     for all $a \in A$ do
11:         if $a$ is a valid action in the current position then
12:             move agent by executing action $a$
13:             $found \leftarrow$ ID-DFS-HELPER$(k, d + 1)$
14:             if $found$ then
15:                 add $a$ at the start of solution $p^\star$          ▷ $p^\star$ is found in reverse order
16:                 return true
17:             else
18:                 backtrack one step in the search tree                 ▷ undo last move
19:     return false
```

4.2 Pruning using Loop detection and Admissible CPP Heuristic

While Algorithm 1 yields an optimal solution, and guarantees it if such a solution exists, it has to analyze many branches of the search tree that are not promising (i.e., have little chance of leading to an optimal solution). Our branch-and-bound planner aims at alleviating this problem by cutting the unpromising parts of the search tree. One type of unpromising subtrees occur when the agent arrives in a grid cell (i, j), which has already been visited, without having covered any other grid cell since its last visit to position (i, j) (i.e., we detected a *loop* in the state-space). In order to take this into account, we introduce the matrix $M = (m_{ij})_{m \times n}$, where m_{ij} is the number of grid cells that remained to be covered the last time the agent was in position (i, j). We modify Algorithm 1 to consider and update this new matrix M. When a new recursive call begins, and the agent is in position (i, j), a condition is inserted to check if $m_{ij} \leq |R|$. If this condition is true, then the current path is clearly suboptimal, and the current subtree is thus pruned from the search space.

A second way to improve Algorithm 1 is to introduce an *admissible heuristic* cost function $h\colon \mathcal{S} \to \mathbb{N}$, i.e., a function that takes as input states $s = (i, j, R)$ from the set of states $\mathcal{S}$ and returns as output a lower bound $h(s)$ on the number of actions needed to cover the remaining uncovered grid cells (the cells in R). Such a heuristic can be used in two ways: (1) It allows pruning even more unpromising subtrees than with the previously mentioned method, and (2) it allows ordering the successors of a state by how much promising they are and thus finding a solution faster by exploring the most promising subtrees first.

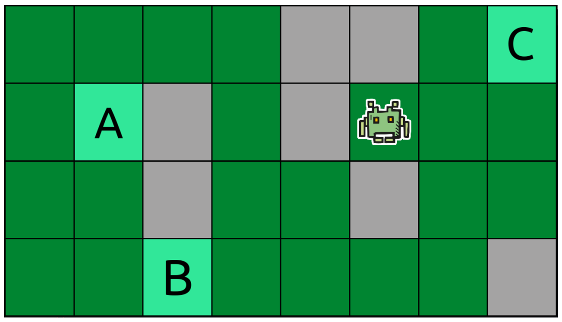


Fig. 2. Example showing how the proposed heuristic works in practice.

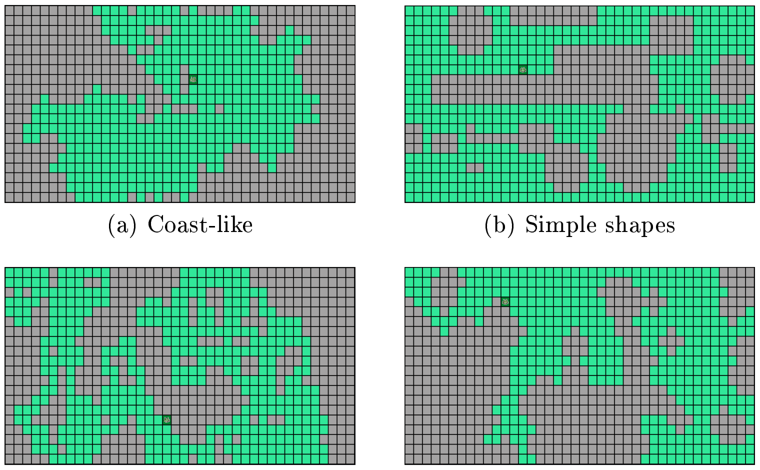
Algorithm 2 Heuristic cost computation

```
1: procedure MIN-REMAINING-MOVES($(i, j, R)$: a state) : positive integer
2:     $left \leftarrow right \leftarrow up \leftarrow down \leftarrow 0$                ▷ Variables initialization
3:     for all $(r_i, r_j) \in R$ do                ▷ Loop on every remaining grid cell to cover
4:         if $r_i < i$ then                ▷ The uncovered cell is above
5:             $up = \max(up, i - r_i)$
6:         else $down = \max(down, r_i - i)$                ▷ The uncovered cell is below
7:         if $r_j < j$ then                ▷ The uncovered cell is to the left
8:             $left = \max(left, j - r_j)$
9:         else $right = \max(right, r_j - j)$                ▷ The uncovered cell is to the right
10:    return $left + right + \min(left, right) + up + down + \min(up, down)$
```

We describe our novel heuristic by explaining how it computes the lower bound using an example presented in Fig. 2. In this figure, three grid cells (A, B and C) remain to be covered. Our heuristic computes the minimum number of every action in $\mathcal{A}$ that need to be done to cover the remaining cells. For example, the action corresponding to "go left" must be done at least $\max(4, 3, 0) = 4$ times and the action corresponding to "go right" must be done at least $\max(0, 0, 2) = 2$ times. Moreover, if the agent goes two cells to the right, the minimum number of moves to the left will now be two more than the previous minimum of 4, i.e., 6. Algorithm 2 shows more precisely how $h(s)$ is computed.

5 Results and analysis

The algorithms described in Section 4 were implemented in `C++`. The tests were carried out on a PC computer equipped with an Intel Core i5 7600k processor and 32GB of RAM (our planner never used more than 10 MB even on the largest grids, thanks to ID-DFS, so the memory usage in not a problem). To measure the performance of our algorithms, we ran each of them 50 times on the same test grids and took the average of the results obtained. Every planner was tested with each of the four kinds of artificial grids shown in Fig. 3. Type (a) grids were generated with the Diamond-Square algorithm and have the shape of a coast [5]. Type (b) grids were generated by randomly placing simple shapes (triangles, discs and rectangles), whereas type (c) grids mimic a random walk on a grid, and type (d) grids include cells with randomly added "links" between neighboring positions on the grid. All grid types were generated with an inaccessible cells density of $(50 \pm 1)\%$.


(a) Coast-like (b) Simple shapes

(c) Random walk (d) Random links

Fig. 3. The four types of generated grids in our benchmark

Table 1 reports the average running times measured for each planner on the considered types of test grids. Every generated grid had a square dimension, shown in column *Size*. The columns L and H, respectively, stand for our two improvements over the ID-DFS (Algorithm 1) planner, i.e., (L)oop detection and (H)euristic pruning. In the table, the character '-' means that the planner failed to solve the problem within 5 minutes. Note that we do not show solutions length in the table since all techniques yield optimal solutions. As we can see, both variants of our algorithm based on the branch-and-bound approach are orders of magnitude faster, on every type of grid, than the ID-DFS planner.

Table 1. Average running times (in ms) required by the ID-DFS planner and the proposed algorithms (Loop detection and Heuristic pruning)

Grid Type	Size	ID-DFS	L	H	L+H
(a)	4x4	0.026	0.019	0.011	0.011
(a)	5x5	178.745	8.36	0.195	0.136
(a)	6x6	-	238154	333.692	97.341
(a)	7x7	-	-	767.201	233.994
(b)	4x4	0.004	0.003	0.002	0.002
(b)	5x5	0.34	0.052	0.016	0.014
(b)	6x6	-	6613.51	28.305	10.739
(b)	7x7	-	-	29249.8	527.177
(c)	4x4	0.01	0.006	0.006	0.006
(c)	5x5	13.498	2.126	0.142	0.1
(c)	6x6	74824	4589.35	22.353	10.841
(c)	7x7	-	-	45515.5	6485.34
(d)	4x4	0.158	0.073	0.017	0.016
(d)	5x5	3.541	0.389	0.058	0.045
(d)	6x6	26947.3	688.076	4.088	1.946
(d)	7x7	-	165167	383.875	70.261

6 Conclusion

This paper considers relevant but not very well studied problem of complete coverage path-planning (CPP). We showed how an exhaustive algorithm based on iterative deepening depth-first search (ID-DFS) can be effectively accelerated using a branch-and-bound approach. The proposed modifications allow the planner to find an optimal CPP solution orders of magnitude faster compared to ID-DFS, which makes it suitable for practical applications.

As future work, we plan to develop and test a method similar to particle swarm optimization (PSO) considering an initial particle that splits the problem into several sub-problems every time there is more than one eligible neighbor. The splitting process will take place until the number of particles reaches a certain threshold N. When this happens, a pruning process destroying the least promising particles can be carried out according to an evaluation heuristic to be determined. We also envisage to use clustering algorithms [9] to decompose a given grid into smaller, mostly independent sub-grids (i.e., similar to cellular decomposition, but for grid environments), which could be covered optimally one by one. Such an algorithm could be also easily parallelized. The use of clustering techniques has helped optimize the computation process in many different fields (see e.g., [2]).

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds de Recherche du Québec — Nature et Technologies (FRQNT). We would also like to thank Alexandre Blondin-Massé, Guillaume Gosset, and the anonymous authors for their useful advices.

References

1. Champagne Gareau, J., Beaudry, É., Makarenkov, V.: An Efficient Electric Vehicle Path-Planner That Considers the Waiting Time. In: Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 389–397. ACM, Chicago (2019). https://doi.org/10.1145/3347146.3359064
2. Champagne Gareau, J., Beaudry, É., Makarenkov, V.: A Fast Electric Vehicle Planner Using Clustering. In: Studies in Classification, Data Analysis, and Knowledge Organization. vol. 5, pp. 17–25. Springer Science and Business Media Deutschland GmbH (2021). https://doi.org/10.1007/978-3-030-60104-1_3
3. Choset, H.: Coverage for robotics - A survey of recent results. Annals of Mathematics and Artificial Intelligence **31**(1-4), 113–126 (2001). https://doi.org/10.1023/A:1016639210559
4. Choset, H., Pignon, P.: Coverage Path Planning: The Boustrophedon Cellular Decomposition. In: Field and Service Robotics, pp. 203–209. Springer (1998)
5. Fournier, A., Fussell, D., Carpenter, L.: Computer Rendering of Stochastic Models. Communications of the ACM **25**(6), 371–384 (1982)
6. Galceran, E., Carreras, M.: A survey on coverage path planning for robotics. Robotics and Autonomous Systems **61**(12), 1258–1276 (2013)
7. Khan, A., Noreen, I., Habib, Z.: On complete coverage path planning algorithms for non-holonomic mobile robots: Survey and challenges. Journal of Information Science and Engineering **33**(1), 101–121 (2017)
8. LaValle, S.M.: Planning algorithms. Cambridge University Press (2006)
9. Mirkin, B.: Clustering for Data Mining. Chapman and Hall/CRC (2005)
10. Mitchell, J.S.: Shortest paths and networks. In: Handbook of Discrete and Computational Geometry, Third Edition, pp. 811–848. Chapman and Hall/CRC (2017)
11. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn. (2009)
12. Shivashankar, V., Jain, R., Kuter, U., Nau, D.: Real-time planning for covering an initially-unknown spatial environment. In: Proceedings of the 24th International Florida Artificial Intelligence Research Society, FLAIRS - 24. pp. 63–68 (2011)
13. Xu, L.: Graph Planning for Environmental Coverage. Thesis, Carnegie Mellon University (2011), `http://cs.cmu.edu/afs/cs/Web/People/lingx/thesis_xu.pdf`
14. Zelinsky, A., Jarvis, R.A., Byrne, J., Yuta, S.: Planning paths of complete coverage of an unstructured environment by a mobile robot. In: Proc. of the Int'l Conference on Advanced Robotics & Mechatronics (ICARM). vol. 13, pp. 533–538 (1993)