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

Using a state-space pruning algorithm with an admissible heuristic function

Jaël Champagne Gareau
Éric Beaudry
Vladimir Makarenkov

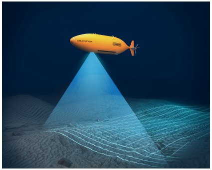
Computer Science Department
Université du Québec à Montréal

November 25–27, 2021

Outline

Coverage Path-Planning

- The **Coverage Path-Planning** (CPP) problem is a motion planning problem, a branch of research that originally comes from robotics.
- Objective : Find a minimal sequence of actions that allows an agent to pass over all points of an area or a volume of interest.
- Applications :
  - robotic vaccum cleaners ;
  - 3d printing ;
  - minesweeping ;
  - underwater autonomous vehicles (AUVs) ;
  - search and rescue ;
  - etc.

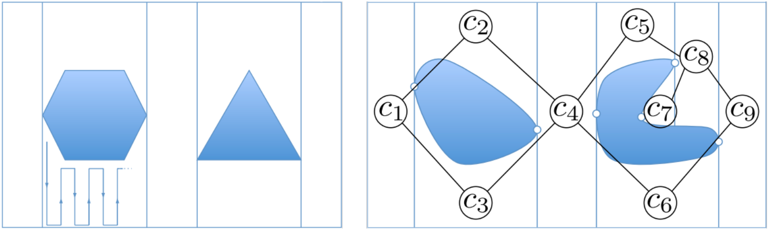


CPP variants

- **Type of environment** : 2D or 3D, discrete or continuous, etc.
- **Allowed movements** : Rectilinear, curved, etc.
- **Type of planner** : offline or online
- **Sensors** : camera, lidar, bumper, etc.
- **Number of agent** : single or cooperative planning

CPP in continuous environments

- Discretize the environment :
  - simple ;
  - can take a lot of memory depending on the resolution.
- Decompose the environment into cells :
  - partitioning of the environement into simple and disjoint regions ;
  - every cell is represented by a node in an adjacency graph ;
  - the problem then becomes :
    1. find a good cell decomposition of the environment ;
    2. find the optimal order of visit of the cells ;
    3. cover every cell with simple movements (e.g., straight lines).

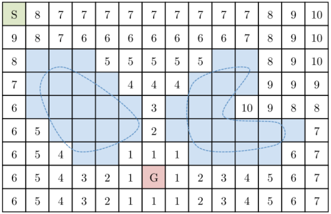


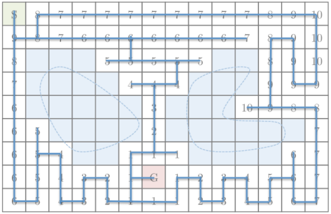
CPP in discrete environments – Representation

- Grid-based representation :
  - simple and contiguous storage in memory ;
  - wavefront algorithm.
- Minimum-Spanning-Tree-based representation :
  - online planning ;
  - the agent cover the environment by following the edges of the tree.
- Graph-based representation :
  - ideal for representating road networks ;
  - can consider environmental constraints ;
  - there is an anytime algorithm.
- Neural-Network-based representation :
  - every cell is a neuron connected to 8 neurons (neighboring cells) ;
  - ideal for unknown or dynamic environments.

CPP in discrete environments – Wavefront algorithm

- Points of departure and arrival are given (they can be the same) ;
- A wave is propagated from the arrival ;
- The agent always visit unexplored neighbors with the highest number first (farthest from the arrival) ;
- No guarantee of optimality ;





Research problem

- Among all CPP planners in the literature, none is optimal in the general case ;
- CPP is an NP-Hard problem : a general optimal solver is $\Omega(b^n)$;
- However, some techniques can be used to improve empirical performance.

Objective

In this research, our objective was to propose, implement and evaluate two ways to increase the computational speed of an optimal discrete CPP solver :

- Branch-and-bound pruning of unpromising subtrees.
- Use of a novel admissible heuristic function.

Problem representation

- The 2D discrete **environment** :
  - is represented by a matrix $G = (g_{ij})_{m \times n}$;
  - g_{ij} indicate if the cell is accessible (to be covered) or inaccessible.
- The **agent** :
  - is the entity doing the coverage of the region ;
  - has a position $p = (i, j)$ on the grid ;
  - can do one of the four actions $\{Up, Down, Left, Right\}$ at each timestep.
- A **state** in the state-space is defined by a tuple $s = (i_s, j_s, R)$, where :
  - (i_s, j_s) is the current position of the agent ;
  - $R = \{(i, j) |$ position (i, j) is accessible and not yet explored$\}$.

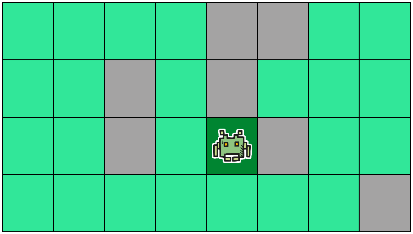


Figure – Example of a CPP discrete environment

Solution to the discrete CPP problem

- An **instance** of the CPP problem is a tuple (G, s_0) where :
  - G is (a matrix representing) an environment ;
  - $s_0 = (i_0, j_0, R_0)$ is the initial state.
- A **solution** to such a CPP instance is :
  - an ordered list of actions (i.e., a plan) $\pi = \langle a_1, a_2, \ldots, a_k \rangle$;
  - the actions move the agent through positions $\langle (i_0, j_0), (i_1, j_1), \ldots, (i_k, j_k) \rangle$;
  - the set of goal states is $\{(i, j, \varnothing) | (i, j)$ is any valid position$\}$

Objective of the optimal CPP problem

Let Π be the set of solutions (plans) of a CPP problem instance. The goal of the CPP problem is to find an **optimal solution** $\pi^\star = \arg\min_{\pi \in \Pi} |\pi|$, i.e., a minimal ordered list of actions leading the agent from the initial state to one of the goal states.

Graph search algorithms

- The state-space can be represented by a graph.
  - Note : the number of states in the graph is exponentially larger than the problem grid.
- Finding an optimal solution of CPP is equivalent to finding a shortest path in the graph from the initial state to a goal state.
- Candidate algorithms :
  - **Breadth-First Search** (BFS) :
    - needs in the worst case to store the complete state-space in memory ;
    - takes too much memory even for very small grids ;
    - algorithms based on BFS (e.g., Dijkstra, $A^{\star}$, etc.) can thus not be used.
  - **Depth-First Search** (DFS) :
    - can go arbitrarily deep in the search tree, even when the solution is close to the root ;
    - can get stuck by expanding the same nodes indefinitely.

Algorithm used in our base planner

- We based our planner on **Iterative Deepening Depth-First Search** (ID-DFS).
- ID-DFS is similar to DFS, but with a depth limit.
- If a solution is not found within depth limit k, DFS is carried out again with a depth limit $k+1$ and continues until a solution is found.
- Ensures the algorithm never goes deeper than necessary and always terminates (if a solution exists).

Base planner

Algorithm CPP planner based on ID-DFS

global
  $\pi^\star$: data-structure (eventually) containing the solution
  $s = (i_s, j_s, R)$: current agent position
procedure ID-DFS-PLAN( )
  for $k \leftarrow |R_0|$ **to** ∞ **do** ▷ k is the depth limit
    found $\leftarrow$ ID-DFS-HELPER$(k, 0)$
    if *found* **then return**
procedure ID-DFS-HELPER(k : depth-limit, d : current depth) : boolean
  if $k = d$ **then return** $|R| = 0$ ▷ returns true iff the grid is fully covered
  for all applicable action a **do**
    move agent by executing action a
    found $\leftarrow$ ID-DFS-HELPER$(k, d + 1)$
    if *found* **then**
      add a at the start of solution $\pi^\star$ ▷ $\pi^\star$ is found in reverse order
      return true
    else
      backtrack one step in the search tree ▷ undo last move
  return false

State-space pruning

- The base planner finds optimal solutions.
- However, it explores some unpromising branches in the search tree.
- By pruning unpromising parts of the search tree, we can greatly improve the planner's performance.

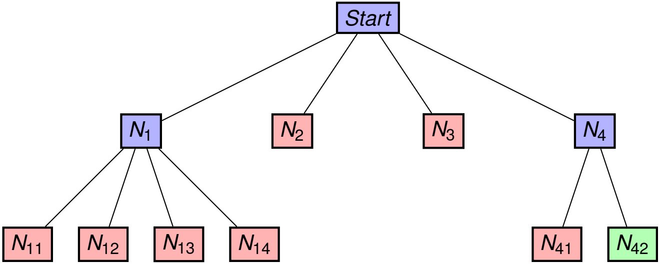


Figure – Example of state-space pruning. Explored, pruned and goal states are respectively blue, red and green.

Loop detection

- One type of unpromising subtree occurs when visiting an already visited cell (i, j) without having explored other cells since last visit to (i, j).
- It manifests as a loop in the state-space.
- Since every action has an opposite action (Up-Down, Left-Right), loops are really frequent.

CPP loop detection with the base planner

- We detect these loops and prune their respective subtrees by :
  - introducing a new matrix $M = (m_{ij})_{m \times n}$;
  - m_{ij} is the number of grid cells that remained to be covered the last time the agent was in position (i, j) ;
  - the base planner is modified to consider and update M ;
  - after every action, if the agent is in position (i, j), a condition checks whether $m_{ij} < |R|$;
  - when the condition is false, a loop is found and the subtree is pruned.

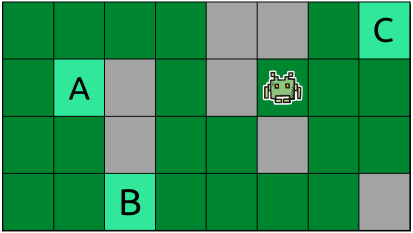
Admissible heuristic

- A **heuristic function** $h\colon \mathcal{S} \rightarrow \mathbb{N}$ is a function that gives an estimate on the cost (number of actions) needed to move from a given state $s \in \mathcal{S}$ to a goal state.
- In AI planning, they are often used to focus a search in promising parts of a state-space and to prune (or ignore) unpromising parts.
- An **admissible heuristic function** is a heuristic function that never overestimates the number of actions needed to reach a goal state.
- There was no heuristic function proposed in the literature for the CPP problem.
- In the CPP problem, such a heuristic function can be used in two ways :
  1. When the number of remaining permitted moves is larger than the minimal number of remaining moves, we know the subtree can be pruned.
  2. The successors of a state can be ordered by how much promising they are (the lower the heuristic value of a successor, the most promising it is).
- Our heuristic function computes the minimal number of times that each of the four actions (go up, go down, go left, go right) need to be used.

Proposed heuristic – Example

Heuristic computation example

- In the figure below, three cells remain to be covered.
- Action "go left" needs to be used at least 4 times to reach A, 3 times to reach B and 0 time to reach C, it must thus be used at least max(4,3,0) = 4 times.
- In total, the number of remaining actions is at least 4 + 2 + 2 + 1 = 9.
- 14 actions are needed to find the optimal solution for this problem.
- We can get a tighter bound by observing that every move in one direction increases the number of required moves in the opposite direction.
- We obtain $h(s) = 4 + 2 + min(4, 2) + 2 + 1 + min(2, 1) = 12$.



Proposed heuristic – Pseudo-code

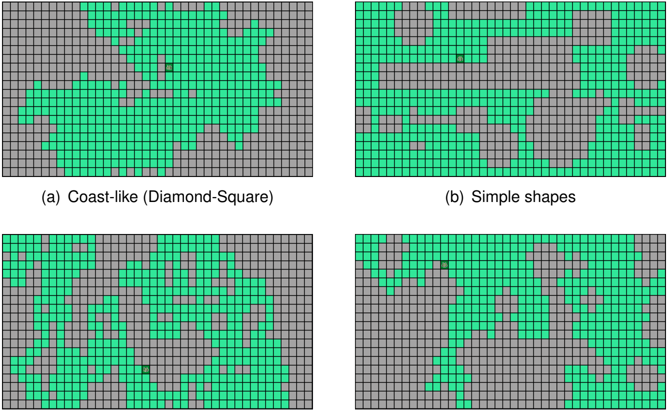
Algorithm Heuristic cost computation

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

Methodology

- We implemented the proposed algorithms in C++.
- The tests were carried out on a PC computer equipped with an Intel Core i5 7600k processor.
- The planner never used more than 10 MB, so memory usage of our proposed planners was not an issue.
- There was no standard set of benchmark environments available in the literature, so we proposed four different types of generated environments.
- To measure the computation performance, we ran each algorithm 50 times on the same test grids and took the median of the obtained results.

Types of generated CPP instances used in our benchmark



(a) Coast-like (Diamond-Square)

(b) Simple shapes

(c) Random walk

(d) Random links

Average running times (in ms) required by the proposed planners

Grid Type	Size	ID-DFS	L	H	L+H
(a)	4x4	0.026	0.019	0.011	0.011
(a)	5x5	178.745	8.360	0.195	0.136
(a)	6x6	-	238154.000	333.692	97.341
(a)	7x7	-	-	767.201	233.994
(b)	4x4	0.004	0.003	0.002	0.002
(b)	5x5	0.340	0.052	0.016	0.014
(b)	6x6	-	6613.510	28.305	10.739
(b)	7x7	-	-	29249.800	527.177
(c)	4x4	0.010	0.006	0.006	0.006
(c)	5x5	13.498	2.126	0.142	0.100
(c)	6x6	74824.000	4589.350	22.353	10.841
(c)	7x7	-	-	45515.500	6485.340
(d)	4x4	0.158	0.073	0.017	0.016
(d)	5x5	3.541	0.389	0.058	0.045
(d)	6x6	26947.300	688.076	4.088	1.946
(d)	7x7	-	165167.000	383.875	70.261

Conclusion

- Optimally solving the discrete grid-based CPP problem is NP-Hard.
- There was no optimal discrete solver described in the literature.
- We proposed a planner based on ID-DFS along with two improvements :
  - a branch-and-bound state-space pruning using loop detection ;
  - an admissible heuristic function allowing pruning and ordering of the subtrees.
- The two proposed improvements lead to orders of magnitude speedup over the ID-DFS planner and can be combined together for further speed improvements.
- As future work, we plan to develop and test :
  - method inspired by particle swarm optimisation (PSO) ;
  - decomposition of the grid using clustering algorithms such that each sub-grid can be solved independantly in parallel.

Acknowledgments

CRSNG NSERC

Fonds de recherche Nature et technologies Québec

We acknowledge the support of the Natural Sciences and Engineering Council of Canada (NSERC) and the Fonds de recherche du Québec — Nature et technologies (FRQNT).