Polytechnic University of Madrid

Msc Artificial Intelligence

Robots Autónomos

Dual Wavefront Algorithm: A Real-time Visualization

[Code]

Miguel Espinosa Miñano

January 21, 2022

Contents

SECTION 1

Introduction

1.1 Abstract

This document intends to be the supporting material for the code implementation of the dual wavefront algorithm.

The code is publicly available in the following github repository: Dual Wavefront Algorithm

1.2 Document structure

The report will:

- describe the algorithm
- comment on the implementation carried out
- give instructions for running locally the project
- provide some final conclusions on the work done

SECTION 2

Algorithm

2.1 Overview

The wavefront algorithm, as it has been implemented in this version, is a graph based algorithm that expands neighbours to construct a tree-based map. The objective is to find a path avoiding the obstacles to move from a *start* node to a *goal* node.

In the dual wavefront version, two fronts simultaneously exist and expand. The two fronts expand, one starting from the *start* node, and the other from the *goal* node. This way, both wavefronts will meet approximately half-way the distance between the start and goal points.

Figure 1 shows how the node expansion that is carried out from both *start* and *goal* nodes.

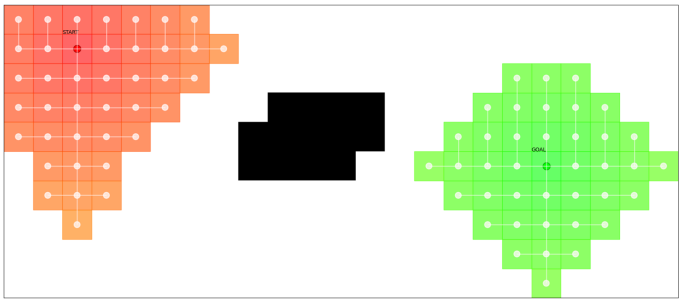


Fig. 1: *Tree node expansion*

Similarly, Figure 2 shows with a dotted line both wavefronts as they expand throughout the search space.

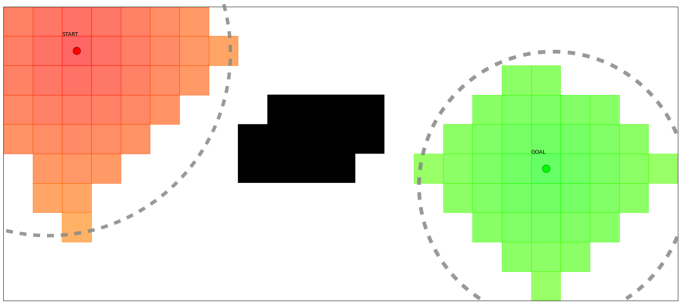


Fig. 2: *Both wavefronts marked with dotted line*

2.2 Description

Before jumping into the pseudocode, it is relevant to mention that the search space is discrete, organised in the form of a grid. Therefore, the implementation of the dual wavefront algorithm is simply a tree expansion search in a discrete space. Also, from the *start* node perspective, there is no prior knowledge of the position of goal, therefore, no heuristics can be used. Similarly, from the *goal* node perspective, there is no knowledge about the position of the start. Therefore, it is a blind search.

Equally, it is worth clarifying that we are making use of trees, and therefore, **each node may have multiple children, but will only have a unique parent**. The way we expand the tree enable us to state that: **each node's parent will always be the most optimal way to go back to the initial root node**.

The dual wavefront algorithm flow is described in the following Pseudocode 3.

Algorithm 1: Dual wave-front algorithm

```
1  init_trees();
2  init_queues();
3  while not meet do
4  |  currentNode ←queue.pop(0)
5  |  children ←neighbours(currentNode)
6  |  for node ∈ children do
7  |  |  currentNode.add_son(node)
8  |  |  queue.append(node)
9  while not home do
10 |  currentNode = currentNode.parent
```

Fig. 3: *Pseudocode for main anti-flocking algorithm*

Following, we explain in more detail each of the functions.

- `init_trees()` (line 1): This function initialises two trees. A first tree with root on the start node, and a second tree with root on the goal node.
- `init_queues()` (line 2): This function initialises two queues: one for the nodes expanded from the *start* node, and another for the nodes expanded from the *goal* node. This queues will contain those nodes that have not been yet expanded. In other words, they contain the leaves of each tree.

- `queue.pop(0)` (line 4): We extract the first node stored in the queue, that is, the node that has been the longest in the queue.
- `neighbours` (line 5): This function will extract all the neighbours or children from a given node that have not yet been expanded, that is, nodes that yet do not belong to any tree.
- `add_son()` (line 7): This function will take a node and add it as son into the existing tree.
- `append()` (line 8): This function appends (or pushes) a *node* into the queue so that it can be later on expanded.
- $currentNode = currentNode.parent$ (line 10): This line performs the backtracking to find the optimal path back to the initial root node.

2.3 Implementation

The implementation of the dual wavefront algorithm has been carried out following the above Pseudocode 3. Four files have been generated.

main.py

This file contains the main execution flow as it has been previously described in the pseudocode shown. In addition, we make extensive use of the library `matplotlib.pyplot` for the graphics visualisation. There is lots of codes for the correct customization, color definition, axis formatting, annotations, lines, dots and rectagles.

functions.py

This file contains auxiliary functions that help modulate the code from the file *main.py*. Among this utilities, the following functions are present:

- `read_file()`: reads a txt with the map definition
- `get_screen_dimensions()`: returns the screen dimensions for the plot axis definition
- `get_plots()`: creates the plots and returns the *figure* and *axis*.

- `gradient_plot()`: computes the gradient plots (this is an extra utility).
- `add_video()`: creates the subprocess handler for the video correct creation
- `eucl_dist()`: computes the euclidean distance between two points in 2D space
- `hexcolor()`: function to return an hex color depending on an input value. This function will enable us to create the gradient color effect present in the animation.
- `add_line()`: function for adding a line to the plot
- `add_dot()`: function for adding a dot to the plot
- `neighbours()`: given a node, return its neighbours, considering only those that are not already present in the current tree structure. **The way the neighbours are computed guarantees optimality for the backtrack process.** That is, each node will only have one parent, and that parent is the optimal one to get back to the tree root.

constants.py

This file contains different constants that will determine different simulation behaviours.

- `VIDEO` (boolean): determines whether a video recording of the simulation should be generated or not
- `ANIMATION` (boolean): determines whether a real-time animation should be launched or not. (Note: animation is necessary for the video creation, as the video is simply taking snapshot frames from the simulation)
- `TREE_EXTENSION_ANIMATION` (boolean): determines whether the tree expansion animation should also be included or not
- `GRADIENT_PLOTS`: determines if the gradient plots should be shown. False by default.
- `VIDEO_NAME`: determines the output video file name

– `INPUT_FILE_NAME`: determines the input *.txt* file name to read the map and obstacles

– `DIAGONALS`: to enable or disable diagonal navigation. (Note that the *diagonals* functionality is not working yet.)

node.py

This file defines the *Node* class and its corresponding attributes. The code with comments is shown below.

```
"""
    Class for tree Node
"""
class Node(object):
    def __init__(self,xy,cost=0,parent=None,root=False) -> None:
        self.xy = xy # Node coordinates
        self.root = root # Boolean: is this node tree root
        self.cost = cost # Cost of reaching this node from root
        self.parent = parent # Pointer to parent Node
        self.children = [] # Pointers to children Nodes

    def add_son(self,son):
        son.parent = self
        self.children.append(son)

    def find(self, xy):
        """ Find node in subtree """
        if self.xy == xy: return self
        for node in self.children:
            n = node.find(xy)
            if n: return n
        return None
```

2.4 Instructions for running the algorithm

To customise the execution, the constants file may be modified to satisfy the user needs.

2.4.1 Custom map creation

The user may define a new map. Maps are simple *.txt* files that need to be formatted in the following way:

- First line should indicate the grid dimension in the following format: `<HEIGHT>x<WIDTH>`
- The following lines dimensions should match the specified *height* and *width* parameters.
- Only one starting point may be placed in the map. To place a start point, the character "`s`" is used.
- Only one goal may be placed in the map. To place the goal, the character "`g`" is used.
- To define obstacles, the character "`o`" can be used.
- The rest of empty space should be filled with characters "`-`".

An example is provided below. Given the input file as shown in Figure 4, the map shown in Figure 5 would be created.

```
10x23
-----------------------
--s--------------------
-----------------------
---------oooo----------
--------ooooo----------
--------oooo------g----
-----------------------
-----------------------
-----------------------
-----------------------
```

Fig. 4: *Example of input file .txt*

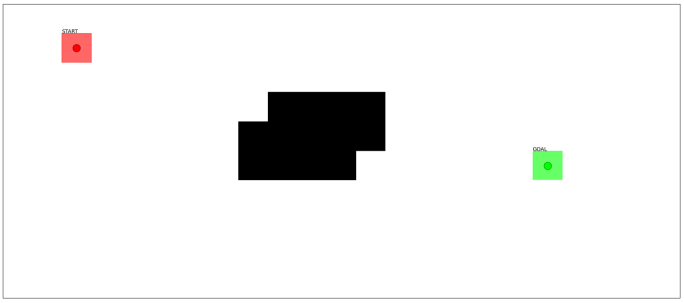


Fig. 5: *Corresponding map generated from input file*

2.4.2 Simulation parameters

At it has been previously explained in the `Constants.py` file, some animation and simulation parameters can be changed according to the user needs.

2.4.3 Packages dependencies

The program requires some external python packages to be installed: `numpy`, `matplotlib`, `scipy`, `tkinter`, `math`, `subprocess`.

2.4.4 Running the program

The program can be ran by simply executing from the command line:

```
$ python3 main.py
```

2.4.5 Maze generator

Finally, a recursive maze generator has been used for the creation of some labyrinth maps. The code is in C++.

Acknowledgements: ***The maze generator has been taken from this github repository [Backtracking Maze Generator], it has NOT been implemented by myself.***

SECTION 3

Experimentation and results

To see a real-time visualizations of the dual wavefront algorithm, please check the github repository: Dual wavefront algorithm

In Figure 6, some snapshots are provided, displaying multiple simulations that have been carried out.

Results obtained are concluded to be satisfactory, having built an intuitive visualization of the dual wavefront algorithm.

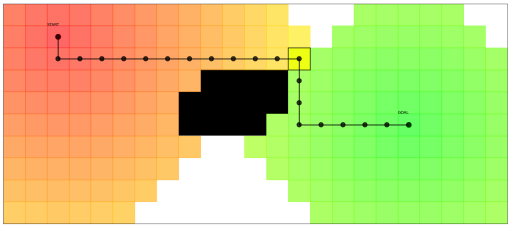
3.1 Discussion

The dual wavefront algorithm has proven to be successful in finding optimal paths from a source location to a target location.

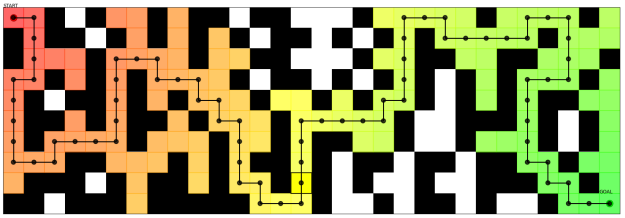
The fact that it is *dual*, (in other words, that we expand simultaneously from both start and goal nodes), makes it twice efficient as the usual wavefront planner.

Given the fact that we are expanding two "wave" fronts, they will always meet at half-way, namely, at the point that is found at the same distance from both *start* and *goal* nodes. Therefore, it will always span the optimal path.

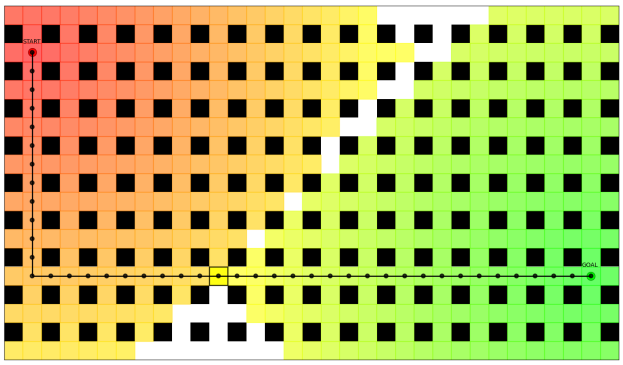
The main disadvantage of this algorithm is the fact that is graph-based. We are dealing with a discrete representation of the search space, which usually does not represent real world scenarios very well.



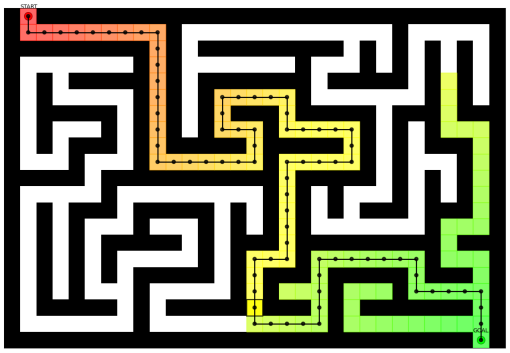
(a) Simple layout



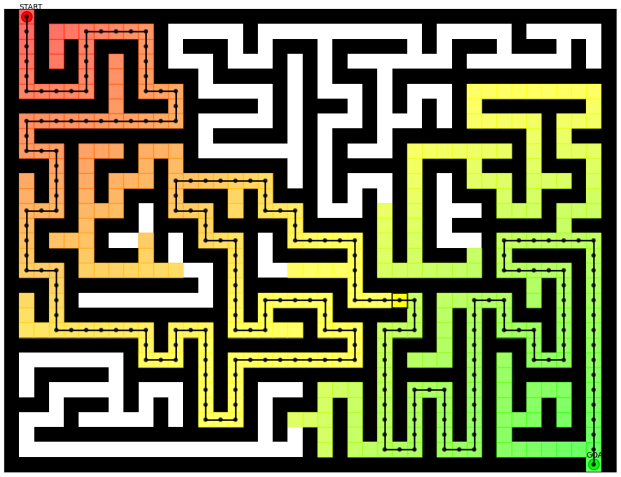
(b) Randomly distributed obstacles



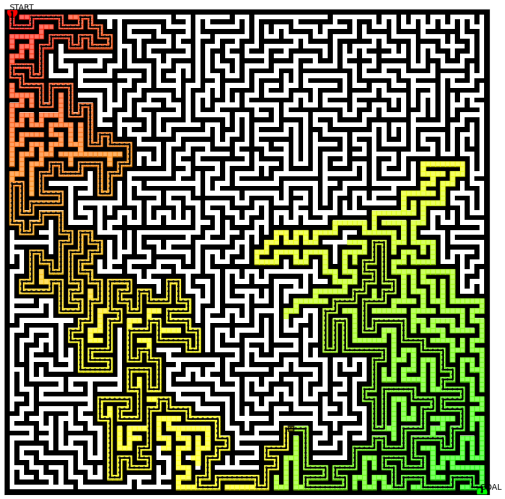
(c) Uniformly distributed dot obstacles



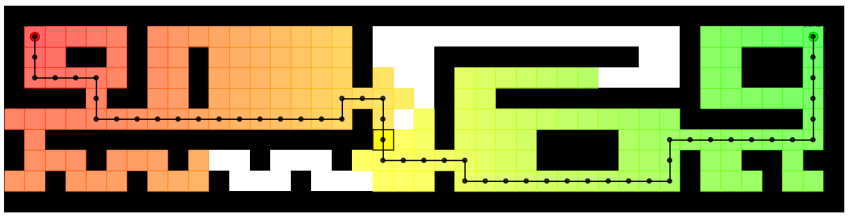
(d) Small labyrinth



(e) Large labyrinth



(f) Huge labyrinth



(e) House map layout

Fig. 6: *Simulation outputs for different map scenarios*

SECTION 4

Conclusions

The following conclusions have been reached:

- The aim of the project has been successfully accomplished by implementing the dual wavefront algorithm.
- The dual wavefront algorithm is more efficient than the basic wavefront algorithm due to the fact that it uses two fronts that simultaneously expand.
- By using a tree structure we are able to directly construct the most optimal path on-the-go, and therefore, once both fronts have met, it is only a matter of backtracking through the parents nodes back to the root.
- We have implemented an intuitive visualisation that helps any user to quickly understand the procedure this algorithm is following. Furthermore, multiple customisation options have been enabled.
- Possible flaws have been discussed, among them the fact that the continuous space is discrete and deterministic.

4.1 Future work

Some future work directions are detailed below:

- As it has already been discussed, the search space is discretized, thus, making the graph search feasible. But, the discrete representation is less realistic. Therefore, one possible future work would be trying out a wavefront algorithm for a continuous space.
- Search space considered here is deterministic. Equally, it could be considered implementing a stochastic dual wavefront algorithm, that is, that adapts to possible changes in the environment. For example, when a robot follows a defined path, deviations and errors may occur. Therefore, dynamically computing new optimal paths (by recomputing the dual wavefronts in real-time) would be interesting.

- A feature that, due to time constraints, it has not been implemented has been enabling the use of diagonals.
- Lastly, trajectories generated are not feasible for a robot (in the strict sense). Therefore, the generation of smoother trajectories via splines interpolation would also be interesting to carry out.